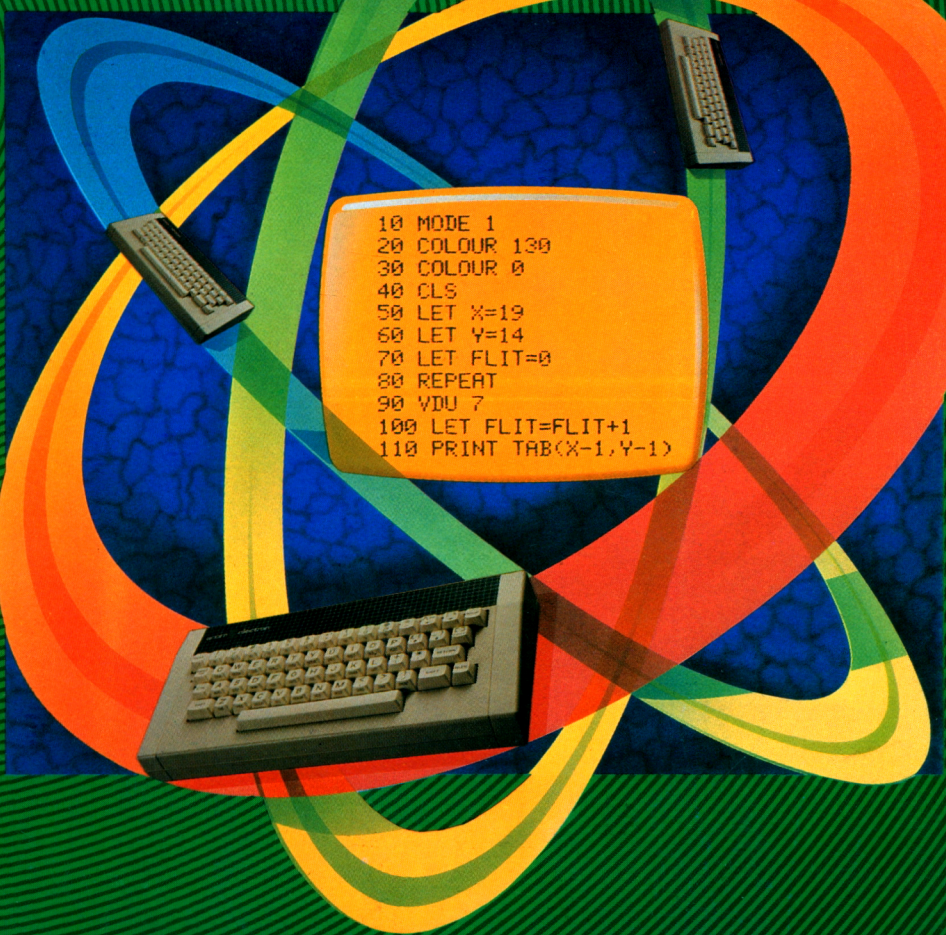
Easy Programming for the Electron

SHIVA's friendly micro series

Eric Deeson



Easy Programming for the Electron

Eric Deeson

Joseph Chamberlain College, Birmingham

Shiva Publishing Limited

SHIVA PUBLISHING LIMITED
4 Church Lane, Nantwich, Cheshire CW5 5RQ, England

© Eric Deeson, 1984

ISBN 0 906812 66 6

Cover photographs, Figure 1.1 and Figure 2.3 of the Electron courtesy of Acorn Computers.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise, without the prior written permission of the Publishers.

This book is sold subject to the Standard Conditions of Sale of Net Books and may not be resold in the UK below the net price given by the Publishers in their current price list.

Typeset and printed by Devon Print Group, Exeter

Contents

	Introduction	1
1	Dive In! (Preparation, Getting it together, Introductory Cassette, Fixtures)	3
2	A Computer? What's That? (Definitions, The parts of a computer)	7
3	What is a Program? (A simple program, Commands and statements, Another program, Loops, Projects)	10
4	What Programming is About (Planning)	16
5	Live On Your Screen (The PRINT statement, Strings, Animation with PRINT, Your own characters, Summary, Projects)	18
6	Feast of Languages (Programming languages, Completely different, Other differences, Editing)	33
7	Join the Colours (FOR . . . NEXT loops, Changing colour, Projects)	37
8	Command Performance (Commands, Editing, Computer memory jargon, Squashing programs, Comments)	44
9	Getting Your Sums Right (Number-crunching, other functions, DIY functions, Projects)	51
10	Getting it Taped (Loading, Saving programs, Loading problems, Error messages)	58
11	Coming to a Decision (IF . . . THEN, More IFs and buts, Conclusion, Projects)	62
12	Picturing Programs (The language of flowcharting, Using flowcharts, Documentation, Projects)	68
13	Sharing the Proceeds (GOSUB . . . RETURN, Procedures, Projects)	74

14 Topping Out Ceremony 84
(Planning, ideas to modules, Modules to flowchart boxes, Coding, Finishing, Structured programming)

15 String Them Up 92
(What is a string?, String operators, String functions, Projects)

16 Bug in a Rug 101
(A bugged program, Tracing, Some closing tips)

17 Sound as a Bell 107
(Sound theory, Electron sound, Synthesis, Envelope, Channel Ø, Projects)

18 Mishmash 114
(True utilities, REM, Mugtrapping, DATA, File-handling, Machine coding, Interfacing)

19 Graphic Phrases 124
(MODEs, The DRAW command, Using PLOT, Trooping the COLOUR, Projects)

20 Hip Hip Array! 132
(Arrays, Projects)

Appendix 1: Extra programs 136

Appendix 2: Notes on questions 147

Appendix 3: Tipples 151

Appendix 4: Resources 160

Index 162

Introduction

A million years ago, well, in early 1980, I horrified my family by ordering a personal computer—Britain's first, the Sinclair ZX80. Things were very different in those days. There was no home computer industry and no-one had heard of Shiva Publishing unless they were post-graduate mathematicians. But to get my own micro was incredibly exciting, even though I had used computers to some extent for a number of years, and had indeed had my first book on the field published over ten years earlier.

Now every High Street is knee-deep in home computer shops and every time you are talking in a group of people, someone's bound to raise the subject of what their own XYZ machine can do. There are thousands of magazines dedicated to the field, and more money changing hands for program cassettes than for music cassettes.

Still, I'm sure you're excited with your Electron, whether it's the first personal micro you'll use or not. And I'm sure that the excitement is tinged with a certain uncertainty, as was mine when I got my ZX80. The uncertainty concerns what effect will the machine have on your life, and on your financial security. And—will you be able to master it like all those brainy youngsters one finds in the computer shops?

Yes—you *will* be able to master it! It's dead easy to get your Electron to carry out at least simple tasks. That's why I wrote this book—to prove that to you, pleasantly and painlessly. Enjoy it, right?

This book is explicitly a guide for beginners to working with the marvellous and popular Acorn Electron microcomputer. I have done my best to keep its style light while trying to explain what can be quite complex concepts. Whether you are using the book at home, at work, or at school, I hope that you will enjoy it, and thus enjoy all the time you spend at the keyboard. Computing is a marvellous hobby and the Electron—like the BBC microcomputer, its bigger, dearer and older sibling—is marvellous. As with all hobbies, once one is fairly deeply involved, progress can be swift and pleasant. It's the first few steps that can be difficult, and I invite you to travel those with me.

The twenty chapters in the book are grouped into several sets. My aim is to help you pick up programming skills and other forms of necessary background knowledge together as you work through towards the end. I don't want you to become just a competent programmer in Electron BASIC—I feel it important too that you gain success in all aspects of what can be a wonderfully engrossing hobby. Please don't skip chapters in order to concentrate on programming—programming is not the only important aim in life!

The Electron *is* a marvellous machine, as I've just said. The marvel lies in the huge range of powerful facilities offered for such a reasonable price. The problem with the machine is that it is designed to expand with your needs and with future developments, and also to be highly compatible with the BBC machine. In order to cater for that expansion and that compatability, programming the Electron (Elk for short) often seems unnecessarily hard. I am sure you will feel that, whether you are quite new to computing or have gained much experience with other machines, cheap or costly.

It is quite true to say that the Elk is not a beginner's machine. This is because of the complexity of its BASIC programming language. Yet experts with other machines have to struggle with the Electron as well, and I think most agree that the struggle is

worthwhile. I have written this book to help minimize *your* struggles. The struggle *is* worthwhile, but it can't be bad to minimize the agony. Enjoy the book—it represents a huge number of struggles on my part—and enjoy and value your new micro too.

This book was written quickly (but not, I hope, hastily) before my first Electron was more than a few weeks old (though I had two more by the time I had finished). It was also before more than a handful of magazine articles had appeared and no books written specifically for it apart from the two you find in the box. To some extent, therefore, I have been working in the dark, and I must express my gratitude to various people for their help. In particular I would like to thank John Coll and Maurice Edmundson of Acorn Computers and Darren James of W H Smith.

I much appreciate too the hard work, allied with a certain resignation, put in by Rita Wagh who faced, with the help of Lynne Nicholls, the difficult task of typing a manuscript of this nature. AVC Software kindly permitted me to use some ideas here from my little book *Learning with the Electron*, and my family have shown their usual patience as I left them alone for many evenings. Soon enough there will be plenty of books on the Electron, I expect. Use this one as initiation, in conjunction with the *User Guide* and programs in the various personal computing magazines. When you have finished this book, I hope it will remain of permanent value for reference, even when you have graduated to a tome on advanced programming.

Birmingham, October 1983 Eric Deeson

DEDICATION

'And still she gazed, and still the wonder grew,
That one small head could carry all he knew.'

I have slightly adapted that nice quote from Oliver Goldsmith as I feel sometimes that it sums up how my mother, Angela Deeson, feels about all I do in computing. This book is dedicated to her with gratitude for her help in keeping my feet on the ground.

Note: A box □ is used to represent a space when one is not otherwise obviously needed.

1 Dive In

Sure, you'd be a most unusual person if you were reading this before getting your Electron microcomputer out of its box and seeing if it works. But we must cater for all tastes . . . Anyway, you may have tried to get going without our magic help and have come across some problems. So, right from the postman's knock, here's how to get stuck into your Elk.

PREPARATION

Prepare your working site You're going to need a TV set, so push the little ones out of the way. Even if they're watching an Open University programme, they've got to learn the new priorities in your home.

You're going to need a power point near the TV, so if necessary get an adaptor (yes, perfectly safe if your system is a 13A ring main). You're going to need a low table for the equipment—clear off Grandma's beer glasses and old newspapers and put it a metre or so in front of the TV.

You're going to need a comfortable seat at that low table. Our kids have got one of those enormous bean-bags—I have priority on that, even if it's murder getting up when the doorbell rings.

If your microcomputer works, and you like using it, prepare yourself for many hours a week at the keyboard. At the end of this section, I'll come back to a proper working site—the above is temporary, but a lot better than working on the floor with the dog treading all over your machine.

Prepare the computer This is what you should have in the box. Keep it all in there when it's not in use. For one thing, bits can get lost, damaged, or attacked by children and other household pe(s)ts if left lying around. For another, computing needs a tidy mind, so we'll start right now. (For another, you may just need to send the computer back; it'll cost you a packet in packaging if you can't find the original stuff.) Anyway it's quite a nice box. I'll start again. This is what you should have in the box:

1. The polystyrene foam container. Note that one half is marked TOP; keep it that way—that will save accidents later.
2. The computer itself. That's the beautiful typewriter thing. Be careful when lifting the machine; it's not quite as light as it looks and the case flexes a bit.
3. The mains adaptor with its neatly coiled long lead. Take a long look at the neat coils—you'll never see them that way again. The adaptor has its own 13A plug already—that saves the usual hassle of initiating new electrical equipment, but does mean that you can't run it with the tape-recorder off one plug. The other end of that long lead ends in a neat little plug that fits into the Electron. It also fits into a Spectrum microcomputer. For goodness' sake, whatever you do, *don't* plug it into your friend's Spectrum. The Spectrum would (I assume) blow up, and so would your friend. Clever trick that, Acorn . . .

4. The video lead. That's the rather short black one. (Black is a bad colour for most of us—hard to see, thus easy to trip over. You *could* get a white carpet I suppose.)
5. The *User Guide*.
6. A free copy of *Start programming* with Masoud Yazdani. I hope you'll be able to understand Masoud by the time you finish with me. (And when you've finished with *him*, maybe you'll be able to wade through the *User Guide*.)
7. An uninspiring-looking paperback-sized box saying 'Introductory Cassette' with, inside it, an audio cassette saying 'Introductory Cassette'. Throw the box away (but keep the cassette until later; it's got some good tunes on it).
8. The guarantee card. Deal with that now before you lose it. You never know . . .

Try to remember where the bits come from. You're going to put them all neatly back again, right? Let me tell you (in case you haven't found out already) that that's not as easy as you might think. That long lead gets everywhere. The box should by now be empty. Oh, except for something a bit like the blue potato crisps we got when I was a kid. Just as well to keep that thing in. I suppose it could be more than a gimmick in a wet winter.

Anything else in your box, such as the odd chip or bug, you are best advised to get in touch with the supplier. Don't forget that messing around inside the micro may not only fail—but will invalidate the guarantee.

Don't be put off by that last paragraph! Chances are that there'll be no problems at all. Anyway it's not my job to write a maintenance manual.

GETTING IT TOGETHER

1. Plug the plug into the socket by the TV. *Don't* be tempted by the 3-metre lead to plug in at the other end of the room—someone is bound to trip over the lead in that case (especially if you actually *do* have a white carpet). Plug the Spectrum-Zapper into the socket on the right edge of the micro (no other hole will take it). The micro will bleep at you now if you used a plug without a switch or when you switch on. At the same time the tiny 'CAPS LK' light on the left side of the keyboard will start shining.
2. Switch on the telly. Don't be distracted by the picture. It's the Open University program the little ones were watching before you turfed them out.

   To get *your* picture:

   (a) Connect the video lead between the 'UHF TV' socket at the back of the micro (do it properly; it's a fairly tight fit) and the TV's aerial socket. (You *must* use a UHF TV, not an ancient VHF one or a less ancient dual-standard set.) The UHF TV socket, clearly marked underneath the micro (!), is the one nearest the back of the set of the four on the left of the case.

   (b) Choose a spare channel on the TV; tough if you watch four channels and have only four switches—use the least popular channel then. You will now see multicoloured or black and white 'snow' on the screen. Depends whether it is a colour or monochrome set you are using.

   (c) Slowly tune the channel's adjuster knob until you get the clear message:

   ```
   Acorn Electron      (Christmas Tree symbol)
       BASIC
       > _     (that dash flashes on and off; the two symbols are 'cursors', or
               prompts)
   ```

   The tuning position is close to the broadcast channels, and very close to those for the Acorn Atom, Sinclair ZX81, Spectrum or BBC microcomputers if you've been using any of those old-fashioned machines before.

3. Turn the TV sound right down. You won't need it while using this book.

4. Type in the following commands. This is where you start learning your way round the keyboard. After each command press the RETURN key; I've coded that (R) here.

MODE 5 (R)	[top two lines of message vanish; rest changes scale]
COLOUR 129 (R)	[your command appears in white; prompts move down, and a patch of red (or grey) appears behind the >]
CLS (R)	[screen clears to red (or grey), + prompts]
PRINT "HALLO" (R)	[use SHIFT key and 2 together to get " and "] [your message appears below the command]

You can play with those commands if you like. I put them in so you could get something on screen to adjust the TV controls. Do that—to get the best picture you can. Adjust whatever your set allows—colour, brightness, contrast, channel tuning and holds. Get the best picture you can; don't worry about the little ones' Open University programme.

If any of this doesn't work and the TV was all right before, either there's something wrong with your power supply (not even step 1 got you anywhere) or with the computer. The latter's not likely really—and certainly nothing I've told you to do could harm the telly.

INTRODUCTORY CASSETTE

Acornsoft or someone have done a lot of work for me in preparing this cassette. Now's the time to use it. Problem: you are going to have to experiment to find out how to connect your cassette recorder (if the little ones haven't got it for an Open University lecture) to the 'CASSETTE' socket at the front left of the micro. Try any audio leads you have in your house—you may be in luck. Otherwise you're going to have to spend time and money getting a suitable lead by post or from a local audio shop. Refer to pages 7 and 8 of the *User Guide*.

Spend a couple of hours or days with this cassette, following Chapter 4 of the *User Guide* and (if you can find them) the relevant bits in *Start Programming*. I'm not suggesting that you spend time on this cassette because it'll put you off my beautiful book, but because it will give you lots of ideas about what micros can do, and will give you some essential keyboard practice. You are not going to get far into using a computer before you'll be needing the cassette recorder a lot—so it's just as well to get it sorted out as soon as you can. Meanwhile, a bit of theory before we go on.

You got the machine to do various things earlier by giving it *commands*. Each had to be completed by pressing the RETURN key (R). This tells the computer that you have finished and it's its turn to do something. There's no way you can get the computer to do a load of things fast by giving it a series of commands. What we do instead is to *program* the computer. This means giving it a series of commands (now called instructions or *statements*); the computer stores these and carries them out when you want.

This computer cannot keep programs stored when it's switched off, so it has the facility to save them as a cassette recording (or in other ways). The Introductory Cassette has a series of such recorded programs; by the time you finish this book, you'll have a series of recorded cassettes of your own, each available for use when you want it.

Now after the theory, a *warning*. When you've worked through this book, you'll be a fairly good programmer. But you *won't* be able to produce programs like some of the more sophisticated ones on the Introductory Cassette. That requires a great understanding of computers and a great experience of advanced programming. And months of hard work sweating at the keyboard.

What I can, and shall, do is later on give you pointers to where you *can* find out more. This book, with the *User Guide* for reference, is only the first step to fame and fortune as a maker of Electron programs. So, let's get on with programming . . . Oops, I nearly forgot. I said I'd come back to a permanent computer user's corner.

FIXTURES

Using a computer can become a marvellous hobby, totally engrossing, totally detrimental to the use of legs and your social life (that's a serious warning really) but as with all hobbies, it needs a bit of effort to fix things up right. You can't hog the home TV set for ever. You can't work for long on a bean-bag. So once you're sure you're going to enjoy computing (if you're not sure yet), *organize*. You'll need the items listed below:

1. A table (at least 2m × 1m) and a comfy upright chair in a quiet corner of a quiet room, with adjacent power, and storage space at hand.
2. A TV set just for this purpose. A black and white portable is OK, but consider a guaranteed second-hand colour set from a local shop. Good ones are available for under £50, and you can secretly watch the midnight movie while you're thinking about your precious program.
3. A permanent arrangement of everything on and around the table, allowing you to work quickly and for a long time without strain. I've sketched a suitable layout for your Electron corner (assuming you're a right-handed user) in Figure 1.1. Make sure it's appealing to visitors when you proudly show off your work—but not too appealing for stray infants poking around in your absence. Or parents . . .

Right, *now* can we start into programming? Actually—no.

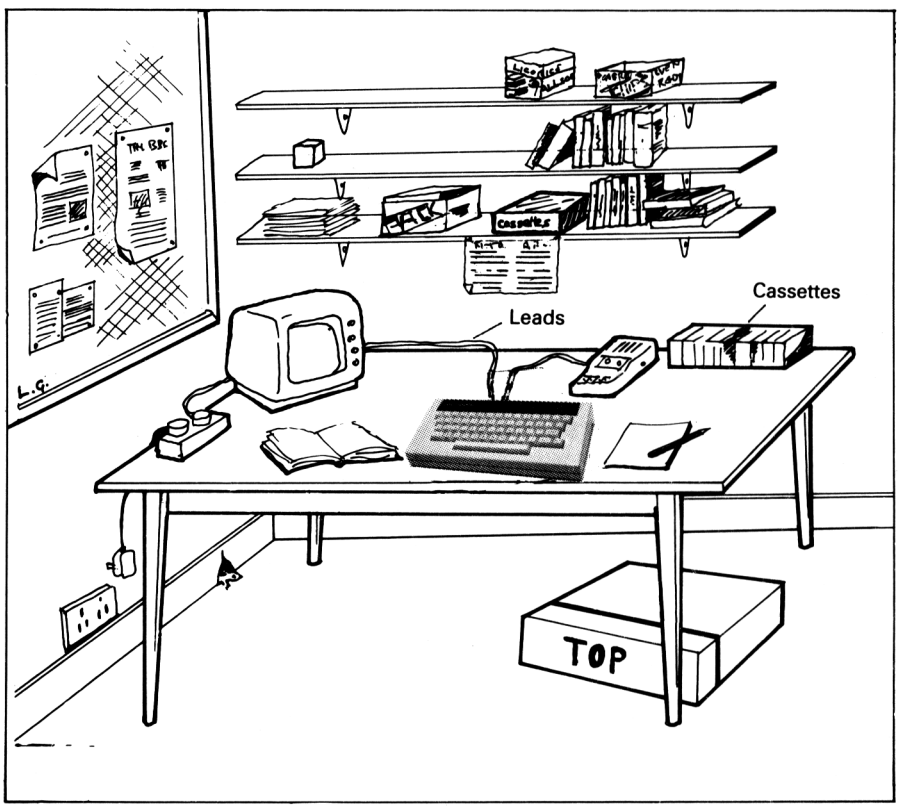


Figure 1.1

2 A Computer? What's That?

'What's the point of this chapter?' you may ask. After all, books on using cars don't waste your time with chapters on what cars are. But please don't skip this stuff. For a start, I've put a lot of work into it, and then—I think it's important. Well, skip it if you like, but read it sometime soon. What *is* a computer?

The first thing to know is, there are lots of kinds of computers. You've probably got some around already—things like digital watches, electronic calculators, video games, posh features in modern cookers and sewing machines. These are all special-purpose computers, *dedicated* to a special task in each case. The Electron, on the other hand, is a *general-purpose* computer—given suitable instructions and attachments, it can do the work of *any* dedicated machine. Perhaps you've run through the 'Introductory Cassette' by now; there are suitable sets of instructions (programs) in there for timing, calculating, and gaming, as well as for various other tasks. To repeat, then, a general-purpose computer like the Electron can do many different things if suitably instructed. A special-purpose computer is dedicated to a small range of functions and cannot (easily) have its instructions changed.

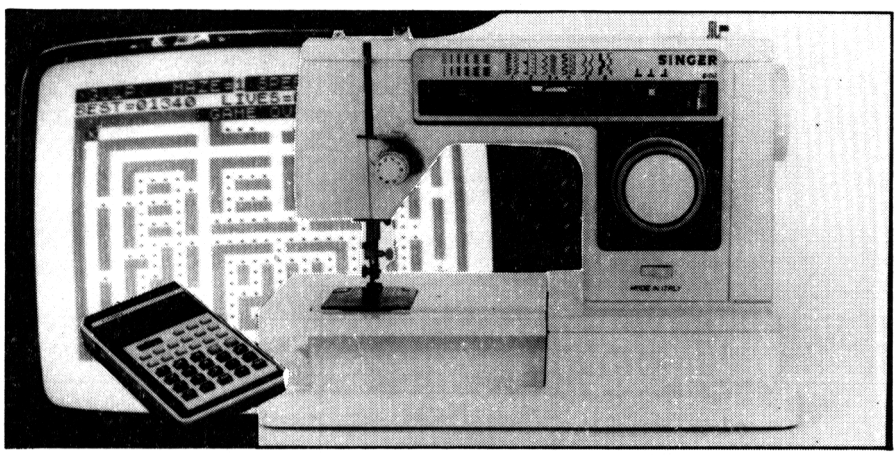


Figure 2.1

DEFINITIONS

Here's what I use as a full definition of a computer like ours.

A modern computer is a general-purpose, high-speed, digital electronic, stored program, data processor.

Phew! I've said what 'general-purpose' means. What about the rest of that description?

High-speed An important thing about computers is that, suitably instructed, they can work much faster than people. (They don't get tired or go on strike either; well, not usually.) I'll come back to this point in a few pages—but modern micros can carry out thousands or millions of actions in a second. That means they can do complex operations very quickly and also keep up with fast changing situations.

Stored program As you may know, a program (*not* 'programme', by the way!) is the name given to a set of instructions for the computer to carry out in order. Now just think what it would be like giving the machine each instruction in turn, when it can carry out millions in a second. Right, the poor thing would be sitting around 99.999% of the time twiddling its electrons while the human operator's getting round to the next instruction. That would be a great waste of the machine's potential, and let's face it, few people would use computers if that had to be done. So the machine must be able to store the instructions somehow, so it can carry them out in a flash on demand. It needs some place to hold them—called a *store* of course—and some way to follow them. The ability to store data and instructions for use on demand is what makes computers so important.

Digital electronic That's a bit technical. It means that this is an electronic system like your TV set, working with tiny electric currents. (Actually it's *micro*-electronic—those currents are *really* tiny.) The 'digital' bit refers to the fact that the currents run around in pulses; these can be counted, so they are like numbers. 'Digits' means 'numbers' (originally 'fingers'). And electronics concerns little bits of matter called electrons. Thus your Electron uses electrons.

Data processor That's what computing's all about. The computer processes data (representing information of interest to humans) to produce more data (more interesting information). Think how computers are used; think how we have already used the Electron—then you can see why some people call computing 'data processing'. A bit like food processing really.

THE PARTS OF A COMPUTER

No, don't worry—I'm not going to get any more technical at this stage. But I think it would help if you understand that all digital computers have the same basic structure. This applies to modern tiny pocket machines, and it applies to the Electron, and it applies to the mammoth monsters the Income Tax people and the Pentagon live with.

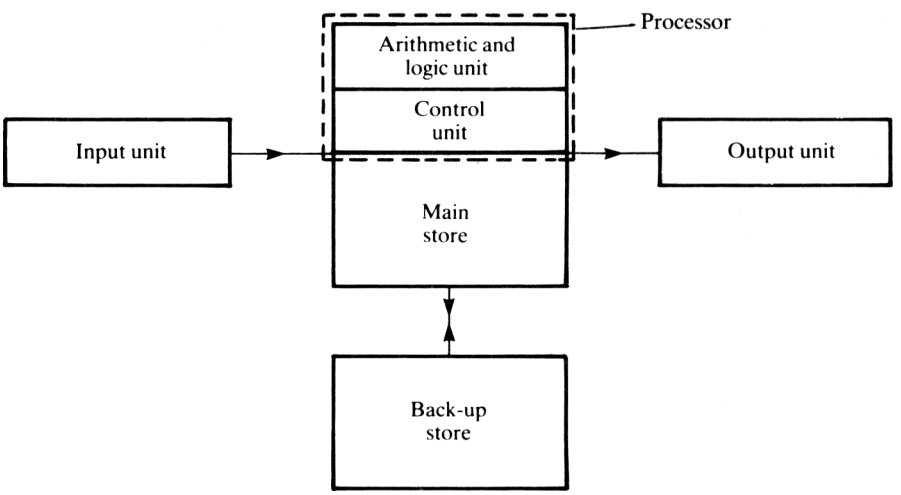


Figure 2.2 The sections of a computer

The bit that does the hard work is called the *processor*. Its job is to process electronic data; that can be thought of as numbers. The part of the processor that deals with this is the *arithmetic and logic unit* (ALU to its friends). Next door is the *control unit* that conducts the whole orchestra.

What does the processor process?—data. How is it instructed?—by programs. Data and programs must be stored. So close to the processor is the memory, or main *store*.

As you learnt from the Introductory Cassette, we can feed material into the micro's store from magnetic tape (and vice versa in fact). So, connected to the main store is the *back-up store*; in our case this is a cassette recorder and cassettes, but there are other kinds around.

All computers must be in communication with the user. Did you see the film 'Tron'? The user has to be able to feed in commands, instructions and data. The *input unit* deals with that need; that's the Electron's keyboard. And the computer has to be able to give the user messages and data—it needs an *output unit*. The TV screen is the standard output unit for micros. A printer, if available, is another one.

The way all those bits fit together is outlined in Figure 2.2. Note that folk often use the word *peripherals* (= outside parts) for the input/output units and back-up store.

There, that wasn't so hard, was it?

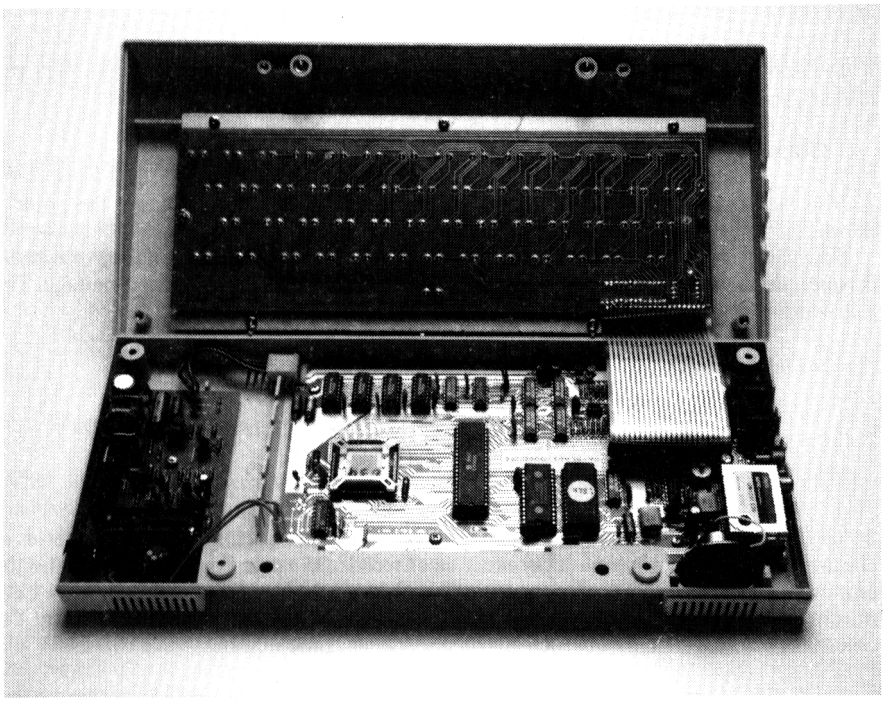


Figure 2.3 "Maybe it's a bit more complex than I've said"

3 What is a Program?

At last we can get stuck into writing programs. So what *is* a program? I hope you find that a simple question to answer, as I've hinted at it twice already!

A program is a series of instructions (called statements) that the computer can remember and follow to do your tasks.

If you want to instruct someone to do something, you must use a language they can follow. Not many of you will (I guess) be able to carry out these instructions:

Odkholl. Min fadlak irsil laenae mayya.

As this isn't a BBC Arabic course I won't go further, but I hope you get the point—computers, like people, must be able to comprehend your instructions.

Your Electron is able to comprehend your program instructions only if they're entered through the keyboard correctly in what's called the BASIC programming language. The following commands that you've already used are written in that language:

MODE 5 (R)
COLOUR 129 (R)
CLS (R)
PRINT "HALLO" (R)

So too is the CHAIN "Intro" that you used with the Introductory Cassette. A series of commands is *not* a program—they must be entered and carried out one by one and the computer doesn't remember them after. To make a program the commands must have numbers in front of them. We call these *line numbers*, and when we use them, the commands take the name instructions or statements. Line numbers have two uses:

1. Each line number tells the micro that the rest is a program instruction, not a command. So it stores it in its memory when it gets RETURN rather than carrying it out at once.
2. The order of line numbers is the order in which the computer must carry out the instructions (unless the instructions tell it otherwise!)

Each program statement must start with a line number (and end with (R)). The line number may be any non-negative whole number up to and including 32767.

I don't expect you will ever write an Electron program with 32767 instructions—a good thing, as the machine's memory isn't large enought to hold that many! So I suggest that in your programs you leave plenty of space between line numbers. I'll stick to line numbers going up in steps of ten as that is common practice. This has a great advantage anyway—it's easy to insert extra lines if you need. If you don't get that—just wait till you *do* need to insert extra statements . . .

COPYING PROGRAMS FROM PRINTED LISTINGS

Most of the material given in this book for you to type into the Electron is in 'hard copy' form. By that I mean it is a copy of the computer's memory made on a printer. Right, the Electron won't yet drive a printer—I had to copy the Electron memory to cassette, using SAVE "name"; copy that copy into my BBC micro, using LOAD "name", and copy *that* with a printer. Anyway, hard copy should be error-free. If your copy in your Electron doesn't work properly, it must be due to a typing error when you entered it. Finding errors isn't easy. Check your screen listing, giving special care to any lines the Electron says are faulty.

Some points for when you transfer hard copy listings into the micro:

1. Take care with Ø and O (zero, sometimes given as Ø, and letter oh), 1 and l (one and little letter ell), 2 and Z, 5 and S, 8 and B, ( ) and < >, : and ; and. and ,.
2. Take care with spaces. I've left lots of spaces in micro material in this book. At first, don't leave *any* of them out except those straight after line numbers. As you get on, *you* can decide which ones to keep in: few are in fact needed apart from the ones in stuff in speech marks.
3. If you get tired on a long program, SAVE it on tape and take a break.
4. If you're on a program using 'assembler' (such material has a line with [at the start, and one with] at the end), SAVE before trying it out in case of catastrophic errors.

All that applies, but even more so, to programs you want to copy that are in ordinary type rather than computer printout form. Especially in magazines . . .

A SIMPLE PROGRAM

Here's a simple program. Enter it. Exactly as it appears here (though the Elk doesn't *need* the spaces, they *do* make the program easier to read.)

```
1Ø MODE 5  (R)                    [Note: Ø, not letter 'Oh"]
2Ø COLOUR 129  (R)
3Ø CLS  (R)
4Ø PRINT "HALLO"  (R)
```

What happens when you've finished entering? *Nothing*. The computer hasn't been *commanded* to do anything, so it doesn't.

To start a program going, we need another command then. It is RUN (R). Try that. Hey presto, it works! You end up with a red screen and the word 'HALLO' (without the speech marks) appearing nicely at the top. Unless your TV is poorly adjusted anyway. Or unless you didn't type the program exactly as printed above. Below the message on the screen are our two friendly prompts. RUN (R) again. I bet you missed what's happened! The whole process took place again, but so fast you can hardly notice it. Try it once more—RUN (R). See? The program works every time you enter RUN (R).

So there's a big difference between commands and program statements—the computer has stored the program instructions and carries them out in order whenever you tell it to RUN (R). But it doesn't store commands—no line numbers to tell it to.

Look, I'll do a deal with you. I don't want to keep saying (R) to you each time you need to press RETURN. Can you remember to use RETURN after each program line, after each command, after anything you've finished typing in? In exchange, here's how to use the RUN printed on the front of the R key:

Every time you hold down the CAPS LK FUNC key and press the R key, the program in store will run. And (I hope not to confuse you) this time you don't even need to press RETURN. In fact, holding down the CAPS LK FUNC key and pressing *any* of the letter

keys will give you the 'key word' printed on the front of that key. As we haven't met most of those keywords yet, I'll not mention that again for the time being.

So, now you've got a program in store. You can prove it by typing RUN (R), or using the CAPS LK FUNC + R keys—every time you do, the same thing happens.

But you never see the program! Well, that's a good thing to the computer user. He or she doesn't want to be bothered by the instructions. However, the programmer *does* want to see what's in store—to change it or check it or just admire it. Use the command LIST. Did anything happen? If not, it's because you forgot (R), didn't you? LIST prints out on a screen a copy of the program in memory. Yes, only a copy—it's still there, as you check with RUN.

COMMANDS AND STATEMENTS

We've seen that BBC BASIC commands—MODE 5, COLOUR 129, CLS and PRINT "HALLO"—can be used in programs as instructions. It that true of them all?

LIST your program and add a new line—50 LIST. RUN that—and you get an *error message*—"Syntax error at line 50". Unlike some micros, the Elk won't accept LIST as a program statement. (That's a pity, actually.)

OK, so try 50 RUN. Now enter RUN again, to run the program. And you get a pretty pattern. Could watch it for hours. The program starts at the start, reaches 50 RUN, so starts again, and so on. This program can't stop. It'll go on for ever. We're in what the jargonists call a closed loop. Could watch it for hours? We're going to have to.

Fortunately, as closed loops may happen by accident, all micro designers have invented ways of breaking out. That's right, the BREAK key. Try it. The pretty pattern stops, the screen clears and we're back to square one. The screen looks almost the same as when the micro was switched on. So have we lost the program? LIST—nothing. RUN—nothing. No, don't fret: we *haven't* lost the program; it's in there somewhere. You *can* get it back—the command is OLD (= get back the old program we thought was lost). Now LIST—there it is. RUN—and off it goes again.

That BREAK OLD (R) routine is a bit of a bind. You've probably noticed the ESCAPE key by now—there it is over on the left. Try it. The program stops with a message referring to a line number. So there's a program with that line number. LIST: OK. RUN: OK. So ESCAPE is a less drastic process than BREAK.

Personally I prefer BREAK, and as that can be programmed to do what I want (within reason anyway), I do so by programming it automatically to provide the OLD and LIST statements that I need. Like this:

```
*KEY 10 "OLD|M LIST|M"
```

[* is SHIFT colon; | is SHIFT with the key next to BREAK.]

This is a useful trick! The fact that you can program the BREAK key in any way you want means that the practical jokers can have a lovely time. I'll leave *that* to your imagination, but see if you can program the BREAK key so that it sets the program running again. That can be useful on occasion. Indeed it is a common method of BREAK-proofing programs, and I'll return to that aspect in due course.

ANOTHER PROGRAM

Time for something different. This next little program is a fairly weedy guess-the-number game—but I shall use it to discuss some basic BASIC statements. And *you* should use it to explore those statements. Before you start, type NEW to clear the memory, and AUTO to let the computer feed you the line numbers. Let me repeat, this *is* a weedy program and it has some faults; you may or may not notice these, but we *shall* be able to improve the program later.

Program 1: Guess-the-number the weedy way

```
10 MODE 5
20 COLOUR 129
30 CLS
40 LET unknown = RND(9)
50 REPEAT
60   PRINT "Give me a number     between 1 and 10."
70   INPUT answer
80   UNTIL answer = unknown
90 PRINT"Right!"
```

Line 10 Spaces optional, remember.
Line 40 SHIFT gives lower case; (on 8 and) on 9.
Line 50 Note the three extra spaces.
Line 90 ESCAPE stops the AUTO routine.

RUN. Does it work? Run and run again. Oh, if the program doesn't work, either I've got to shoot myself for poor proof-reading—or you should shoot yourself for poor typing.

Use BREAK to give you a listing, and follow through the comments below.

1. MODE (line 10): the Electron offers seven different display modes (see the *User Guide* and later in this book). Of those MODE 5 is a colour mode with large characters, as you have seen. Personally I use it quite often. Try different modes, by entering different line 10s in turn and running the program. Compare the effects. You will find only MODE 2 giving effective results apart from 5. Well, maybe MODE 1 too. That depends on your criteria of effectiveness . . .
2. COLOUR (line 20): this statement is used to select background colour (in MODE 5: 128 black 129 red 130 yellow 131 white) and foreground (text) colour (0, 1, 2, and 3 correspondingly). Explore the effects of different colour statements in MODE 5 and in the other modes. Note which ones you like best! MODE 2 is the most colourful.
3. CLS (line 30): the screen clears to the background colour and prompts.
4. LET unknown = RND(10) (line 40): we call this an *assignment statement*. It means that the computer must choose at random—using RND—a whole number between 1 and 10 (inclusive) and call it 'unknown'. If you explore this random thing, remember to change the numbers in line 60 as well. (Thus if you use LET unknown = 10 + RND (20), the limits in line 60 would be 11 and 30.) In this kind of assignment statement, the name before the = (the assignment operator) is called a *numerical variable* (a number that can vary in value). Its value is assigned to be the value of the *numerical expression* after the =.

LET	unknown	=	20 + RND (30)
instruction	numerical variable	assignment operator	numerical expression

I find it helpful to pronounce = as 'become'. Strictly we should say 'take the value of', and the back-arrow used in Figure 3.1 would be a better symbol for it.

LOOPS

REPEAT . . . UNTIL . . . (lines 50 and 80) define the start and end of what folk call a *loop*—the micro is told by REPEAT to carry out the following instructions again and again UNTIL something is true. I can show you best what is happening here with a diagram; there it is in Figure 3.1.

Can you see what's happening? Each time through the loop the computer PRINTs on screen the message 'Give me a number . . .'. It then displays a prompt, the query mark, telling the user to do something, and waits till the user types in a number and (R). The computer then assigns the input number to the variable 'answer' and checks whether 'answer' = 'unknown' or not. If not—round the loop again.

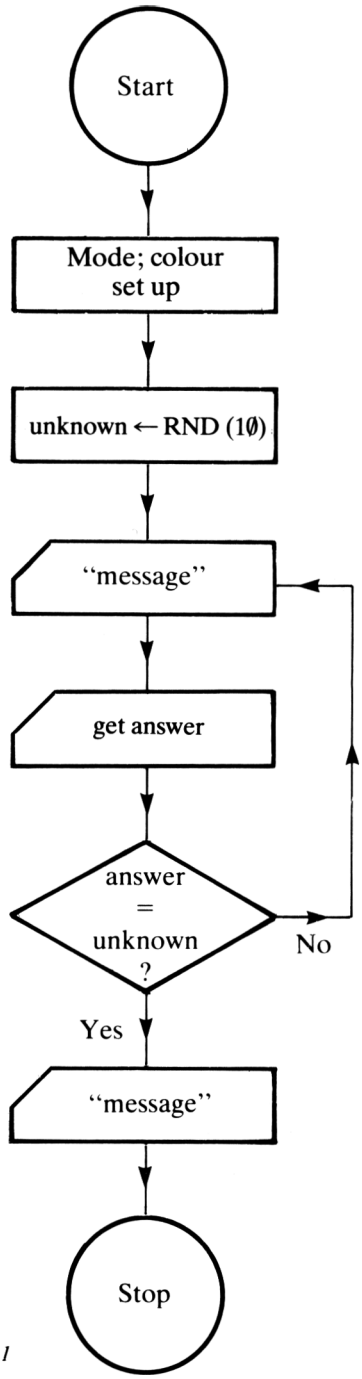

Start
Mode; colour set up
unknown ← RND (1Ø)
"message"
get answer
answer = unknown ?
No
Yes
"message"
Stop

Figure 3.1

Much of that has of course told you about the INPUT statement. It corresponds in a way to the input part of a computer discussed a few pages back, with PRINT corresponding to the output side. INPUT is quite complex. To repeat, 'INPUT ans' tells the micro to:

1. Display the prompt, '?'.
2. Wait for a keyboard input + (R).
3. Print on screen what was input.
4. Assign the input number to the variable 'ans'.

And that's pretty well dissected Program 1. If you want to relax a lot, enter and play with the first program (A1) in the set at the end of the book. But try to understand it too! If you want to relax a little, try some of the projects below.

However, just before you go, I think I had better check the use of the CAPS LK and SHIFT keys as we're beginning to need to understand them. There'll be more on this at the end of the next chapter, but meanwhile:

1. When the CAPS LK light is off, the keys work just as on an ordinary typewriter—lower case (small) letters, numbers, and the symbols on the lower half of the keytops. *Unless* you hold the SHIFT key down. Then you get upper case letters and the symbols on the upper half of the keytops. (If the upper half of the key carries two symbols we get the one on the left.)
2. When the CAPS LK light is on, you get upper case (capital) letters, numbers and the symbols on the lower half of the keytops *UNLESS* you hold SHIFT down. Then you get lower case letters and the symbols on the upper (left) part of the keys.

Confusing? Maybe! But actually there's neat logic in there somewhere, and it's worth sussing out. If you don't, you'll make lots of annoying little mistakes. One question; well, two really—

How do you get the CAPS LK light off? Hold down SHIFT and press the CAPS LK key. How to get it on again? Just the same.

Now practise, and *then* relax.

PROJECTS

(Please refer to Appendix 2 for possible solutions.)

1. Adapt Program 1 by each of these stages in turn:
   (a) Let the program re-start automatically each time the user gets the number right, until "answer" = Ø. (Ø is called a 'rogue value'—one used to stop a loop.)
   (b) Insert 25 COLOUR 2, and RUN to check the effect. Then try to arrange things at lines 2Ø and 25 so that background and foreground colours are assigned at random—but are never the same (why not?).
   (c) Introduce a line 75 starting IF answer < > unknown THEN PRINT something suitable. Note that '< >' means 'doesn't equal'.
   (d) Improve the other PRINT messages.
2. Use the symbols < (is less than) and > (is greater than) with IF and messages as in Question 1(c), to give the user clues in that Program 1.
3. Add to Program 1 the line 45 LET guess = Ø and work out how to count how many guesses the user has had. Then tell him or her.
4. Make up a program of your own using REPEAT/PRINT/INPUT. Make this so good that the rest of your family enjoy running it. Don't forget to NEW first! (Why?)
5. Throughout, get used to the keyboard, including the entry of upper and lower case letters (capitals and little ones), SHIFT and FUNC.

4 What Programming is About

This book is *not* a programming text, as you will know if you are unusual enough to have read the Introduction. Sure, I am trying to help you get into the hobby of programming your Electron, but I am *not* going to spend a lot of ink on telling you what good programming is (even if I were immodest enough). When you've finished the book, and perhaps started to think seriously of programming for work, real fun, or profit, then *you* must find out what makes a good program.

All the same, there are good habits and bad habits. And if you get into good habits early on, you will be glad later. And then your work, fun, or profit-making programming will be less work, more fun, and—who knows?—more profitable.

PLANNING

The essence of good habits in programming is *planning*. Maybe I should go further and say that good programmers plan their planning. The worst thing to do when you get a program idea (whether trivial or ambitious) is to sit down at the keyboard and start typing. If you do that YOU WILL GET INTO AN AWFUL MESS. And even if, by a miracle, the result seems to work first time, it will not be very efficient; it may have BUGS in it (dreaded hidden errors); and it will not impress the folk who look at it and use it.

Efficiency in programming means having a product which works correctly in all circumstances, runs at the right speed, uses a minimum of memory, and looks good to everyone involved. So your planning must aim for these criteria. It'll still be a miracle if the thing is perfect the first time you try it, but getting close to perfection will be a lot simpler. The essence of program planning is an approach called *top-down development*. This means breaking that initial idea down into smaller and smaller chunks until at last each one is an easy to handle unit. Actually we use the words *module* or *routine* for those final units. It's a bit like having a bar of chocolate at a party. You've got to break it up (down?) into the right number of pieces of the right size before it can be processed. Here the processing I'm thinking of is producing the idea in the form of a program the computer can run, *coding* it as we say.

Coding is simple when you have the idea in fragments in front of you. Each fragment —concept or module—can then transfer to a chunk of the program. The chunk is called a *routine* (or a sub-routine to confuse you), or a *procedure*. The sketch in Figure 4.1 shows what top-down programming entails. See what I mean? The original broad concept breaks down into a lot of small chunks. You may also be able to get the idea from Figure 14.3 on Page 89. *That* kind of program description is called a *flowchart*, and I'll have more to say about it later.

Now I am not going to say that you have got to work out a set of modules for the short programs we're into at the moment. But I *do* say that you should view your program not as a whole but as a number of separate parts. Each part has its own aim; each part is programmed (coded) separately and tested separately. Then the whole thing, made of those parts added together, is likely to be neat, efficient, *and* bug-free.

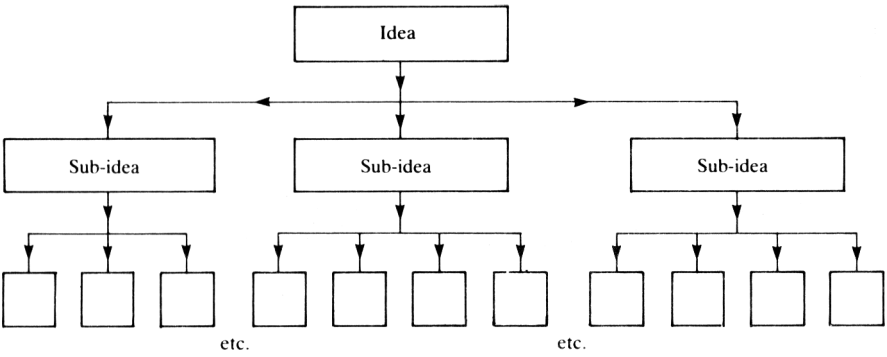


Figure 4.1

As well as working correctly and efficiently, the program must also appeal to the user in other ways. Plan from the outset to make your work *user-friendly* so that the user is always at ease and never wonders what to do. The most important thing here concerns planning the screen messages. These should be short but clear, well laid out and not cramped together—and *make sure that your spelling and punctuation are correct*! (I am a teacher and am too often horrified by finding poor grammar in teaching programs—that to me is inexcusable. Some researcher recently pointed out, by the way, that it is better for a programmer to be good at English than good at maths; this is partly why, surely.)

I have already started using large and small letters in screen messages in this book. These are easier to read than capitals alone and also make the screen more interesting. Get to know the CAPS LocK key therefore. When its light is on, typing a letter gives you the capital (upper case) form, unless you press SHIFT; when the CAPS LK light is off, you get the lower case except when using SHIFT. Complicated? Practise it. After all, messages in capitals alone are *dreary*.

There is a catch here, though. All the keywords you use—PRINT, LIST, CHAIN, CLS and so on—*must* be in upper case letters (capitals) or the computer won't recognise them. That means it's safest to re-press CAPS LK after typing in lower case material. That's a bind, I know—but the result *is* worthwhile. Not quite the same as ordinary typing, eh?

5 Live On Your Screen

Surely one of the most versatile BASIC instructions is PRINT, which sends an output to the screen display. We've used it a lot already. In this chapter I want us to go through with care the different ways of using it. We shall look at PRINT instructions to:

1. Produce a nice-looking display;
2. Put your own shapes on screen;
3. Move characters around the screen to give animated displays.

Maybe you think the last two are the most important. However, please cast your mind back to Program 1, and to the improvements you carried out (didn't you?) for the exercises in Chapter 3. Recall how cramped and messy was the screen. Not good, that. So, using PRINT to make a neat, clear display *is* crucial.

First, I'd like you to BREAK and, if necessary, define the BREAK key to give BREAK OLD (R) LIST (R) as before. (This assumes you've just switched on. If you haven't, type NEW as well, to clear the memory.) Perhaps I'd better remind you how to do that programming BREAK key bit. Type in the following, followed by RETURN:

```
*KEY 10 "OLD|M LIST|M"
```

Note that strictly the speech marks aren't necessary in that key-programming command, but I think they do help to make the meaning clearer. The same is in fact also true of all the spaces in that line, and I guess you can start leaving things like this out once you are sure about what you're doing.

Now enter (using AUTO), RUN (using the FUNC key + R), and think (using your brain) about this little program.

```
10 MODE 5                          [colour mode]
20 COLOUR 129                      [red background]
30 COLOUR 2                        [yellow foreground]
40 CLS                             [clear screen]
50 PRINT "Some message or other."
```

If you've entered that exactly as shown, you should have (at least) two criticisms about the display. The first is that the message is cramped almost unreadably at the top of the screen. Indeed, if your TV (Electron?) is like mine, you will find the top line partly hidden by the frame of the tube! A second problem is that the message appears on the top line except for 'r.'—the message is broken in the middle of a word because in MODE 5 you can have only twenty characters (numbers, letters, spaces etc) on a line. There are many ways of using PRINT, however, some of which will help you get over the first problem. While we're dealing with that, notice how the second problem can be dealt with too.

THE PRINT STATEMENT

Unless the computer is specifically told otherwise, each time it meets a PRINT instruction, it starts a new line. This is true whatever it is going to print, even if it's nothing. So "PRINT nothing" means skip a line. Try it—BREAK and add line 45 PRINT to your program. Better—yes?

That's one way we can get over our first problem. The second way is to use the TAB function, which is rather like the TABULATE you get on posh typewriters. Remove line 45 by entering 45(R)—you can check that with LIST—and enter this new version of line 50:

```
50 PRINT TAB(24); "Some message or other."
```

When you RUN this version of the program—lo and behold, we've got over *both* our display problems. Not only is the message started on the second line, but it is indented so that the last word is no longer broken.

How does TAB (number) work then? What it does it to count off the number of places given in the brackets before it starts displaying the message. Check that—RUN and count the number of spaces before the message appears.

Here's a slightly posher PRINT program now. First use AUTO (FUNC key with A) to get the line numbers for free, then enter the program, exactly as listed, and RUN (FUNC + R). I hope you get no error messages . . .

Program 2: A simple name pattern

```
 10 MODE 5
 20 COLOUR 129
 30 COLOUR 3
 40 CLS
 50 PRINT
 60 PRINT TAB(22); "What is your name";
 70 INPUT name$
 80 LET tab = 1
 90 REPEAT
100   PRINT
110   PRINT TAB(tab) name$
120   LET tab = tab + 1
130   UNTIL tab = 10
```

Line 60 Note final semi-colon.
Line 70 Note final $ sign.
Line 120 Counting.

RUN several times; criticise and think. Practise, again, use of upper and lower case (large and small letters) when giving a name. And why not try other responses than your name? I admit there are problems if what is entered is much longer than FREDERICA. OK, I know that that program is not as brilliant as Elk Invaders, but we *are* getting on, and anyway its function is to help with TAB.

A formatting keyword rather similar to TAB is SPC. Indeed some folk have stated that SPC is the Electron's equivalent to the TAB of some other machines. That is, I think, confusing, if not downright wrong.

If you're still in MODE5, COLOUR 129, COLOUR 2 try the following. Enter COLOUR 130 (R), to give a yellow background to new material. Then use these two commands:

```
PRINT TAB (10) "TAB"          [brackets essential]
PRINT SPC (10) "SPC"          [brackets not essential]
```

SPC prints the specified number of spaces before the "; TAB jumps over that number of places. It won't always show this difference, but I suggest you be strict—use TAB for

moving the PRINT position; use SPC only when you actually want a row of spaces.

In fact the above program illustrates several points about PRINT. Here they are.

1. PRINT TAB(number), as in line 60—to start printing at a suitable point on a suitable line. Here TAB(22) means go 22 positions on (from the top left of the screen)—to the third point on the second line in MODE 5.
2. PRINT TAB(variable), as in line 110—to start printing at a point determined by the value of a 'variable'. Here that variable is cleverly named 'tab'.
3. PRINT nothing, as in lines 50 and 100, to skip a line.

The program also introduces two new ideas. One is the dollar sign ($), which I'll come back to in a bit; the other is the matter of punctuation, as with the semi-colon at the end of line 60. I've said before that folk call BASIC a programming language, and as you already realize it has its own special vocabulary. It has its own special punctuation as well, and the semi-colon used a couple of times in Program 2 (line 60) is an example.

Normally when the computer comes across PRINT, it will print the next thing on a new line. After the computer has dealt with line 60, it turns to line 70 and, as we know, one of the things it must do when it meets INPUT is to print a question mark. Did you notice that when the program was run, the question "What is your name" was followed by a question mark in the right place, although there wasn't one given within the speech marks? In fact, it's that little tiny semi-colon at the end of line 60 that has caused this to happen. Normally the question mark of the INPUT statement would have appeared on a new line; the semi-colon before that prevented this from happening.

Most versions of BASIC require you to have semi-colons separating the different items that need to be printed as part of a single PRINT statement. Thus most computers would expect you to type line 110 like this—PRINT TAB(tab); name$. You can compare this with what I did in line 60 to see what I mean. The Electron does not require a semi-colon in this kind of situation, so normally I shall leave it out, and suggest you do the same.

The last line of the next program asks the computer to print a number of 'print items' all in a single statement. I've put a couple of semi-colons in there but, as you will find if you try, they are not actually necessary. With the use of punctuation marks like the semi-colon, PRINT statements can be very flexible. The next program shows this, and also introduces the apostrophe (')—this tells the computer to go to the next line. Does that sound silly after what I said about the semi-colon? Try the program and see. Use NEW before you enter, and don't forget AUTO. By the way, if you don't realize it, you get the apostrophe using SHIFT + 7.

Program 3: Address format

```
 10 MODE 5
 20 COLOUR 130
 30 COLOUR 1
 40 CLS
 50 PRINT TAB(22) "What is your name";
 60 INPUT name$
 70 PRINT TAB(22) "And your house andstreet";
 80 INPUT street$
 90 PRINT TAB(22) "And the place";
100 INPUT place$
110 COLOUR 0
120 CLS
130 PRINT'' name$'' TAB(2); street$'' TAB(4); place$'''''
```

Line 70 Space missed on purpose.

I guess you found out about the DELETE key by now. It's the one on the bottom right corner that lets you wipe out the last character you typed. Of course you can also use it when entering stuff in a program run, and you may well find it necessary in this case.

Did you come across a problem when entering your house and street? It may be that you put in a comma, as many people do when doing that part of the address. You will have found out, then, that INPUT does not accept the comma in the middle of what you type in (the reason may become clearer later), and the best thing you can do in a case like this is just omit any such commas. (If you really need to enter strings of characters which include a comma, you must put the whole lot within speech marks.)

The important thing in Program 3 is line 13Ø. Please make sure you understand exactly how it lays out ('formats', to use the jargon) the screen displays the way it does. Remember that the semi-colons I've used there are not essential, but the apostrophes are.

I've used CLS (= clear screen) about a million times by now. I expect you realise that it too is of great value for screen formatting. I suggest you get into the habit of clearing the screen often, to remove finished-with material, to prevent clutter—to make the display as nice and useful as you can.

There's one more PRINT punctuation technique to know about—using the comma. It appears in line 13Ø of Program 4 (which, as usual, also employs one or two other new ideas!).

Program 4: Tables of squares

```
  10 MODE 5
  20 COLOUR 128
  30 COLOUR 2
  40 CLS
  50 PRINT' "Give me a number!"'
  60 INPUT number
  70 PRINT'' "Thanks!"
  80 CLS
  90 PRINT'' TAB(4) "Number" TAB(14) "Square"' TAB(4) "====
==" TAB(14) "======"
 100 LET newnumber = number
 110 REPEAT
 120   PRINT
 130   PRINT newnumber, newnumber^2
 140   LET newnumber = newnumber+1
 150   UNTIL newnumber = number+9
```

Line 7Ø Nice to be polite, though I bet you don't see this in the run!
Line 13Ø ^ , the 'power' sign; comes with SHIFT on the left-arrow key.

The comma in micro PRINT statements corresponds to the restricted TABULATE you get on cheaper typewriters like the one I use. In effect the screen display is divided into what folk call *zones*, each 10 character spaces wide. As the mode we've lately been using, MODE 5, has 20 characters on a line, it has two zones, but in fact most of the others in the Electron have four. PRINT, "message" will start the display at the beginning of the second zone. Numbers, on the other hand, appear lined up at the right of the zone used, as you saw with line 13Ø of Program 4, unless you tried excessively large numbers with it.

STRINGS

Now (at last) I can get back to the bit I saved after listing Program 2, the bit about the symbol $. Program 2 was the first time we met this—where line 7Ø is INPUT name$, and line 11Ø is PRINT . . . name$.

The symbol $ stands for, and is pronounced, 'string'. So what's a string? Well—look at that line 7Ø again: INPUT name$. We learned earlier that when the Electron is told to input something, it displays a prompt (?), waits for a keyboard entry + (R), assigns what was entered to a variable, and then goes on. When we first met INPUT, it concerned the keyboard entry of a number. Now it concerns the keyboard entry of a string—*any* set of keyboard characters.

All the messages we've used are strings; so too are the answers to questions like "And your house and street?" Strings can contain numbers, as your street address probably does, but they are *not* numbers—even if they contain nothing but numbers. If that's complicated—don't worry; I'll come back to it in detail later.

So INPUT name$ is how we get a string into a running program. And PRINT will display strings on screen for us. In fact, as you've seen, PRINT can be used to display a string in one of two ways. In the case of PRINT "your message here", the string "your message here" is a *string constant* (an unchanging string). In the case of PRINT name$, name$ is a *string variable* whose value must have been assigned to it before. So we can have:

```
10 PRINT "What is your name";          [print string constant]
20 INPUT name$                         [assign to string variable]
30 PRINT name$                         [print string variable]
```

or even

```
10 LET recommendation$ = "Watch Horizon!"   [assign constant to string
                                             variable]
20 PRINT recommendation$                    [print it]
```

Try the next program, and see some of these points in practice.

Program 5: Favourite colour

```
  10 MODE 5
  20 COLOUR 129
  30 COLOUR 3
  40 LET mycolour$ = "PINK"
  50 LET message$ = "What colour do you like best"
  60 REPEAT
  70   CLS
  80   PRINT'''' TAB(1) message$;
  90   INPUT answer$
 100   UNTIL answer$ = mycolour$
 110 PRINT''''' mycolour$; " is my favouritetoo."'''''' "G
lad we agree...."''''''
```

Line 40 Change this if you want!

Dig the proliferation of apostrophes (line 110). And dig the improvement to screen layout that they produce.

Let's extend the TAB idea a bit. After all, I guess you'll guess it can be a bit fiddly as used at the moment. We've seen how you can PRINT TAB(number) to reach any point in any line. Thus in MODE 5, PRINT TAB(122) "A" would put A in the position shown in Figure 5.1, while PRINT TAB(222) "B" would put B in a position 222 spaces from the start. However, to put a character in any position this way is both clumsy and hard to use. Also, its result would depend on what else had already been printed. (Try, in MODE 5, PRINT TAB(122) "A" TAB(222) "B", for instance.)

Fortunately we can also use TAB(x, y) to print directly at position 'x' on line 'y'. Thus PRINT TAB(2, 22) "C" would give the result for C shown below.

Try the next little program out, and study it carefully. It is, by the way, the first one (I think) that you may possibly find of any permanent use. See Chapter 10 if you want to save it on tape. I also give more detailed notes after the listing than usual.

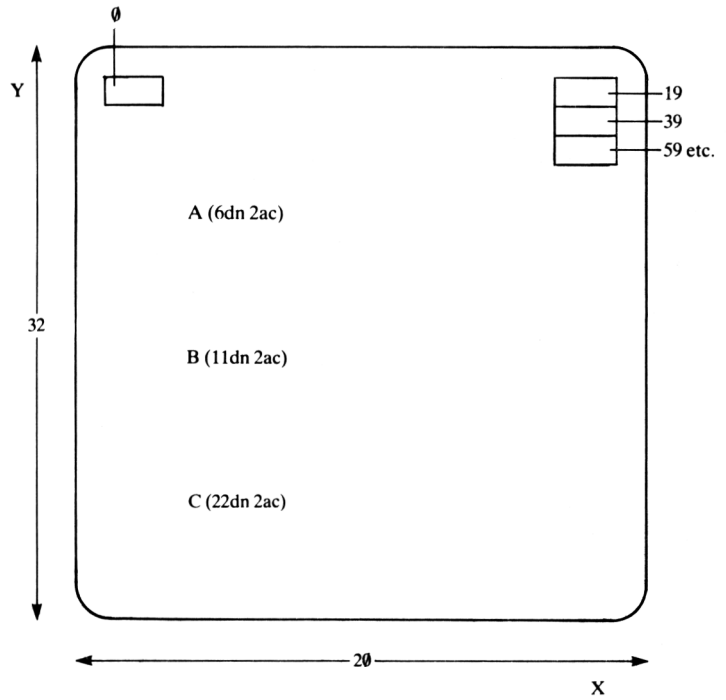


Figure 5.1

Program 6: TABula rasa

```
 10 MODE 5
 20 COLOUR 131
 30 COLOUR 0
 40 CLS
 50 REPEAT
 60   PRINT TAB(2,2) "What line";
 70   INPUT line
 80   PRINT "  What column";
 90   INPUT column
100   PRINT "  What character";
110   INPUT character$
120   PRINT TAB(2,2) "                "' "                    "'
"                    " TAB(column,line) character$
130   UNTIL character$ = "Z"
140 PRINT TAB(0,31);
150 REPEAT
160   UNTIL FALSE
```

Notes

Lines 20 and 30	You may have had enough experience with COLOUR by now to change these lines if you prefer.
Line 120	The empty string constants here are to wipe out the previous messages.
Line 130	Here is a *rogue value* again—the program will run round and round the REPEAT loop UNTIL the character entered is Z. The program will then print Z at the position you have just indicated and proceed to line 140.

Lines 150 and 160 This is a useful trick to stop a program from stopping, if you see what I mean. There is no way of getting out, except with use of ESCAPE or BREAK.

When you use this program you will be asked to indicate the position at which you would like a specified character printed. You may then build up some sort of 'picture' to your heart's content, until you get fed up enough to type Z (the usual indication of boredom). Note that as the program will print that Z on the screen, you should choose where you would like it to be and indicate that position in the usual way.

I repeat that Chapter 10 will give you some notes on how to save this (or any other) program on tape; alas, I wish I could tell you how to print the beautiful pictures you make onto paper—but that is not easy with the Electron, even if you have a printer. Without a printer, it's even harder.

If you did try this carefully, you will have noticed some strange things. Thus remarkable effects follow PRINT TABing outside the screen limits—MODE 5 has 32 lines of 20 spaces (and other modes have different values, as you can see from the table on Page 31).

But the most important thing to note is that the first line is not number 1 but number 0, and the first column is likewise number 0. As a result, PRINT TAB(0, 0) puts you at the top left corner of the screen, and the bottom right is position (19, 31) rather than the (20, 32) you may have expected.

Now let's try to do something more interesting with TAB . . .

ANIMATION WITH PRINT

Walt Disney, here we come! Here's a game (of sorts). Enter it, try it—and do your best to work out what is happening at each point. Again this program may be worth saving.

Program 7: Flit

```
  10 MODE 1
  20 COLOUR 130
  30 COLOUR 0
  40 CLS
  50 LET x = 19
  60 LET y = 14
  70 LET flit = 0
  80 REPEAT
  90   VDU 7
 100   LET flit = flit+1
 110   PRINT TAB(x-1,y-1) "   " TAB(x-1,y) " * " TAB(x-1,y+
1) "   "
 120   LET rand = RND(2)
 130   IF rand = 1 THEN LET x = x-1
 140   IF rand = 2 THEN LET x = x+1
 150   LET rand = RND(2)
 160   IF rand = 1 THEN LET y = y-1
 170   IF rand = 2 THEN LET y = y+1
 180   FOR wait = 1 TO 250: NEXT
 190   UNTIL x = 0 OR x = 39 OR y = 0 OR y = 31
 200 PRINT TAB(0,9) "Fly lasted ";flit " flits."
```

Line 90 Beep from speaker.
Line 100 Counts flits.
Line 180 A 'delay' loop.

Note the use of random numbers, embodied in the 'rand' variables from line 120. And note too that just for a change we're working in a different mode. MODE 1 is pretty much the same as MODE 5, but it allows more characters on a line. That means the

program lasts longer. You may not think that is a good thing, in which case change line 10 to MODE 5, the value in line 50 to 9 and the upper value of 'x' in line 190 to 19.

Don't worry about the 'delay' loop in line 180—its purpose is to slow the 'fly' down but not to slow *you* down—we'll find out later how it works.

Scientists would call this a 'random walk' program. Who cares? It is a sort of game for our purposes, but I agree that it is non-interactive, which means that all you can do is sit and watch it. I suppose that in fact you could get together with a gang and place bets on which side of the screen the fly will leave. Principally this program is an exercise in TAB, so be particularly sure you understand lines 120–170 and 200.

YOUR OWN CHARACTERS

Actually the Elk star (*) that I used in the last program makes quite a nice fly. But this particular computer allows you to design and use your own characters if that is your pleasure.

A tiny bit of theory first. Computers like the Elk can have up to 256 different characters in their *character set*. These carry numbers from 0 to 255. Yes, again the first on the list is called zero. 256 characters is a lot. Add up all the ones you can think of:

Upper case letters (capitals)	26
Lower case letters	26
Digits (0–9)	10
Punctuation signs	11
Mathematical ones	9
Other symbols (£ etc)	3 or so
Space	1

Total—a lot less than 100!

Now, to use the facility of 256 possible chunks, some computers offer lots of special characters, called graphics blocks; examples are card suit symbols and shapes for building pictures. Other computers put the keywords (IF, PRINT, TAB, INPUT, and so on) in as single characters. Others again completely waste the potential.

The Electron does none of these, but allows you to reserve some characters for your own use whenever you please. (Indeed you can use more than the 32 officially available, but the methods concerned there are too advanced for us.)

Say you want a little snake. A tildier tilde than the Elk's tilde (the sign you get with SHIFT + CTRL + left-arrow key). All you need to do is tell the Elk to call your snake character number 224 (say). Then you get it when you want by asking the machine to print the character coded 224: PRINT CHR$ 224.

Be warned—to define your shapes is a bit lengthy. But it is worth it. Here is what you do.

1. You need to sketch your shape on an 8 × 8 grid. Use old graph paper or pages from a square ruled exercise book, or feel welcome to make copies of the grids in Figure 5.2.
2. Sketch your design faintly on the 8 × 8 grid. A sketch for our snake is shown in Figure 5.3.
3. Reproduce the shape as close to that sketch as you can by blacking in the right squares of the grid. Figure 5.4 is my attempt.
4. Now—the tricky bit. Refer to Figure 5.5 to make it easier . . . For each of the eight lines of eight squares in turn—write down the sum of the numbers at the tops of the columns containing blacked-in squares. First line—the only black square is 64, so the sum is 64; second line—black for squares 128 + 32, giving 160; third line 16 + 1, giving 17; and so on. I think you had better check my arithmetic!
5. Enter 10 VDU 23, 224, 64, 160, 17, 17, 17, 17, 10, 4. Here we are defining, using the magic key VDU 23, the character number 224 to be coded with those eight numbers.

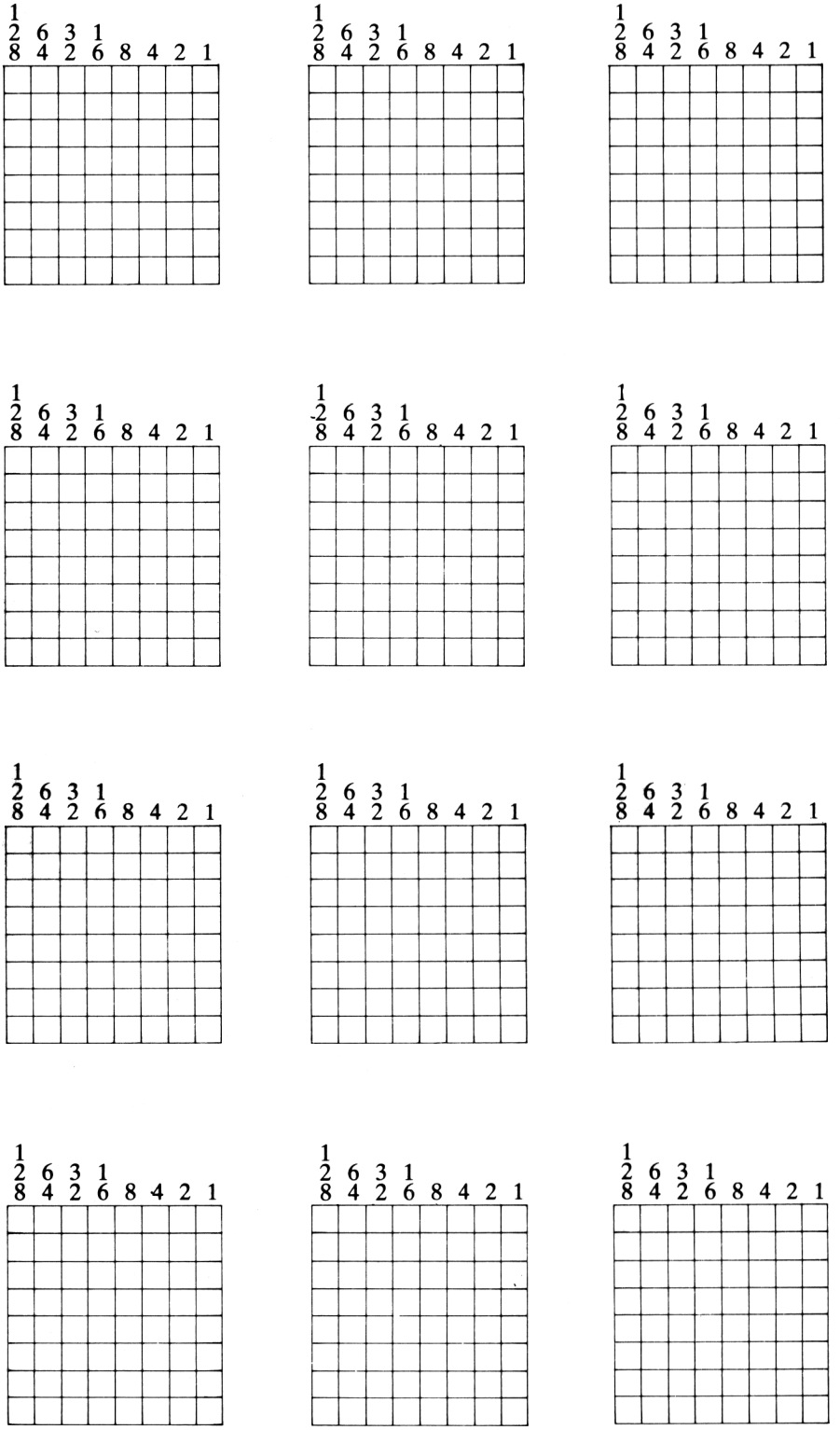


Figure 5.2

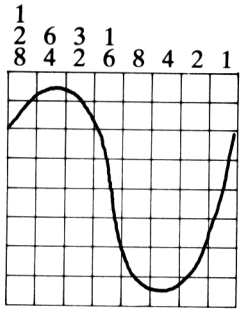


Figure 5.3

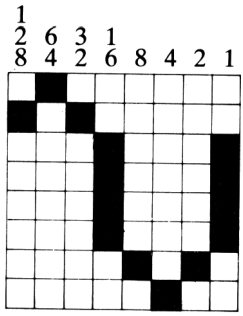
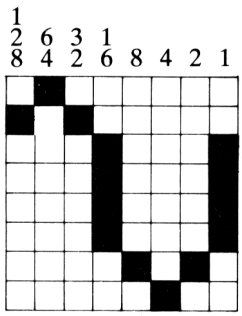


Figure 5.4



64	64
128+ 32	160
16 + 1	17
16 + 1	17
16 + 1	17
16 + 1	17
8 + 2	10
4	4

Figure 5.5

Here's the rest of a little program that would test the first line. Enter and RUN it.

```
20 MODE 5
30 COLOUR 129
40 COLOUR 2
50 CLS
60 PRINT TAB(10, 15) CHR$ 224
```

OK—see what's happened? We've got a nice little snake in the centre of the screen. If you look at it closely you can just about see it's made up of dots like those I sketched.

Was it worth it? All that, just to get a snake in the centre of the screen. Ah, well . . . now the graphic design world is your oyster! Here are a few ideas to get you going.

Our little snake suddenly becomes a wavelet. Make big waves by using PRINT STRING$(100, CHR$ 224). (Stay in MODE 5 for this.) Here I've used a new keyword, STRING$(number, string). This can be very useful for stringing together little strings to make big strings, if you see what I mean. The number in brackets after the keyword tells you how many substrings to stick together, and the substring concerned is the second thing you enter in the brackets. I think you'll be able to suss that out if you check the wave example above.

Another thing you can do is to try Program 8—RUN it and sit back and relax by the sea.

Program 8: Sea-change

```
10 VDU 23, 224, 64,160,17,17,17,17,10,4
20 MODE 2
30 COLOUR 130
```

```
40 COLOUR 11
50 CLS
60 REPEAT
70   PRINT TAB(RND(20)-1,RND(32)-1) CHR$ 224
80   UNTIL FALSE
```

Notes

Line 10	No spaces are required in this statement, but as usual I have put extra ones in to make the line more readable.
Lines 20–40	Just for a change we'll go into MODE 2, where more colour is readily available. Refer to the table on Page 141 of the *User Guide* for the details (if you can follow them).
Line 70	Check out with care how I've combined TAB and RND.

Of course you can try your own shapes, giving them character numbers between 224 and 255 inclusive. (You *can* use others, but they will replace existing shapes.)

For instance, the sketch in Figure 5.6 and the program below it could give us a better fly for Program 7.

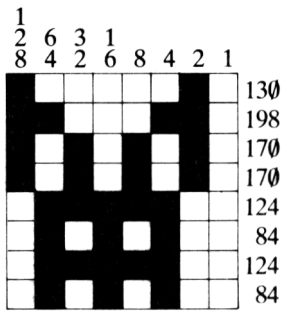


Figure 5.6

```
1  VDU 23, 225, 130, 198, 170, 170, 124, 84, 124, 84
10 MODE 5
20 COLOUR 129
30 COLOUR 0
40 CLS
50 (etc. as before (with print CHR$ 225 instead of "*" in line 110) )
```

And here's a couple more you can try if you want:

```
VDU 23, 226, 32, 112, 82, 60, 60, 36, 108, 0
VDU 23, 227, 36, 90, 165, 90, 60, 155, 24, 60
```

Now to achieve some sort of culmination, why not try your own Space Invader shape? I bet a few thousand other people have done that already!

You can make bigger designs, if you like, by (literally) stringing shapes together. We've already seen the use of STRING$ for that purpose. Now I'll give you another way. Here's the front half of a pantomime horse:

```
VDU 23, 225, 96, 172, 255, 223, 31, 40, 68, 130
```

Here's the back half:

```
VDU 23, 227, 0, 14, 250, 250, 248, 40, 68, 130
```

And here's the whole thing:

```
LET horse$ = CHR$ 225 + CHR$ 227
```

PRINT horse$ in any mode (except 3 and 6) and see that lovely horse. Lovely? Well, I'm no artist. It's worth noting, by the way, that once a character is programmed like this, it stays in computer memory until you re-program it or switch off the micro. That means you can RUN VDU 23 . . . lines and then delete those lines from the program listing.

SUMMARY

PRINT followed by nothing—skip a line
PRINT followed by something—start printing on next line
PRINT' something—start on next line but one
PRINT; something—leave no space (often you can leave the semi-colon out)
PRINT, something—print in next zone
PRINT TAB(n) something—jump n places then print
PRINT TAB(m, n) something—start printing at line n, column m
PRINT SPC n something—print n spaces then something

And the somethings?

PRINT numeric constant, e.g. PRINT 4 gives 4
PRINT numeric variable, e.g. PRINT numeric gives 4 if 'numeric' has the value 4
PRINT string constant, e.g. PRINT "A" gives A
PRINT string variable, e.g. PRINT string$ gives * if string$ has the value "*"
PRINT CHR$ n prints the character whose code is n

PRINT is a versatile instruction indeed!

Now—it is up to you. We've looked at animation with PRINT, and you've got the hang, I hope, of programming your own characters—so, what can you do now?

If you're short of ideas, try some of the projects. Perhaps you should try them anyway. Or you should be able to follow most of Program A2, Train race at the back of the book.

PROJECTS

1. Prepare a grid of small squares (using graph paper or squared exercise-book paper is a good idea), 20 across and 32 down. Plot on it a chart of the stars of your Zodiac sign. Write a program in MODE 5 with black paper (background), and yellow ink (foreground) to print out the starfield. Use * for each star, of course, and TAB(x, y) and TAB(x) for print positions.
2. With the same approach, write a program to display a picture of your choice. Study the keyboard characters, and experiment with them, to find which are most useful for 'graphics blocks'. Program 6 can give a start, perhaps.
3. Enter, run and study Program 9. Make sure you understand (a) how it handles strings, (b) how it TABs, and (c) how I've used VDU 23.

Program 9: Favourite food

```
10 MODE 5
20 COLOUR 130
```

```
   30 COLOUR 1
   40 VDU 23, 224, 170,85,170,85,170,85,170,85
   41 REM  **  useful character this!  **
   50 LET a$ = "Carrots"
   60 LET b$ = "Baked beans"
   70 LET c$ = "Tomato"
   80 LET d$ = "Tomato ketchup"
   90 LET x$ = CHR$ 224
   91 REM  **  to make less work later  **
  100 CLS
  110 PRINT' "Your favourite food?==================="
  111 REM  ** underline trick  **
  119 REM  **  start of 'menu'  **
  120 PRINT TAB(2,10) "1. " a$
  130 PRINT TAB(22) "2. " b$
  140  PRINT' TAB(2) "3. " c$;
  150 PRINT TAB(42) "4. " d$
  151 REM  **  end of 'menu'  **
  152 REM  **  NB different TAB techniques
  160 PRINT''''''
  170  FOR wait = 1 TO 1000: NEXT wait
  171 REM  **  delay  **
  180 PRINT "Please enter number, and RETURN."''
  190 INPUT choice
  200 IF choice<1 OR choice>4 THEN GOTO 190
  201 REM  **  simple error-trap  **
  210 CLS
  220 PRINT TAB(1,15) "Here you are, then!"
  230 FOR wait = 1 TO 2000: NEXT wait
  231 REM  **  delay  **
  240 CLS
  250 GOTO (200+choice*100)
  251 REM  **  a 'computed' GO TO - useful          **  See l
ater note  **
  300 PRINT TAB(5,5) x$' TAB(4) x$ x$ x$' TAB(4) x$ x$ x$' T
AB(4) x$ x$ x$' TAB(5) x$
  301 REM  **  there ARE easier ways!  **
  310 PRINT TAB(15,15) x$' TAB(14) x$ x$ x$' TAB(14) x$ x$ x
$' TAB(14) x$ x$ x$' TAB(15) x$
  320 PRINT TAB(2,20) a$
  330 FOR wait = 1 TO 2000: NEXT wait
  340 GOTO 100
  341 REM  **  re-start  **
  400 LET go = 1
  409 REM  **  in SOME books the next four lines counts as a
 whole program, called MEASLES  **

  410 REPEAT
  420   PRINT TAB(RND(19),RND(31)) x$
  430   LET go = go+1
  440   UNTIL go = 50
  450 PRINT TAB(2) b$
  460 FOR wait = 1 TO 2000: NEXT wait
  470 GOTO 100
  500 PRINT'''''' TAB(9) x$
  510 PRINT TAB(8) x$ x$ x$
  520 PRINT TAB(7) x$ x$ x$ x$ x$' TAB(8) x$ x$ x$ TAB(9) x$

  530 PRINT'' c$
  540 FOR wait = 1 TO 2000: NEXT wait
  550 GOTO 100
  560 PRINT'''''
  570 PRINT TAB(12) x$' TAB(11) x$ x$ x$ TAB(11) x$ x$ x$ TA
B(11) x$ x$ x$ TAB(11) x$ x$ x$ TAB(11) x$ x$ x$
  600  PRINT'''''
  610  PRINT TAB(12) x$' TAB(11) x$ x$ x$ TAB(11) x$ x$ x$ T
AB(11) x$ x$ x$ TAB(11) x$ x$ x$ TAB(11) x$ x$ x$
  620  PRINT TAB(202) d$
  630  FOR wait = 1 TO 2000: NEXT wait
  640  GOTO 100
```

Notes

This is our first long program, so that a listing of it takes up (much) more than a single screen. Before you type LIST, hold down the CTRL key and press N. This will cause the listing to come in 'pages', each new page being accessed with SHIFT. Use ESCAPE and CTRL + O to escape from this form of listing should you so wish.

The next new thing I have introduced here is the use of 'remarks', the relevant keyword being REM. When you enter the program don't feel you've got to enter all those REMs. They are there to give comments to you and they will be ignored by the computer so won't affect the running of the program. On the other hand if you wish to record the program it may be helpful to include the REMs, so that when you later examine the listing you will remember what's going on. The use of REMark is most important. Get into the habit of it with your own programs so that you will understand after some period what you have been doing. I would suggest that you follow the approach used in Program 9, giving REMarks line numbers ending with 9 or 1. I have gone even further and put a load of spaces at the end of each so that they stand out more in the listing. You may not feel that that is worthwhile, and cetainly it takes up more of the computer's meagre memory.

The next thing introduced in this program is the use of the keyword GOTO. There are many people who will scream with blue fits if they see a GOTO in a program, for they believe that it is a clear symptom of bad programming practice. However, GOTO has many valuable uses that cannot otherwise readily be produced, and this program shows one (in line 25Ø). (You may be able to avoid GOTOs with careful use of what we shall later call subroutines or procedures.) At the moment, don't worry. This program works well and is properly structured. I should have said this before! To get out of the above program, use ESCAPE (or BREAK).

4. Now practise your character-defining technique!
   (a) Rewrite the program of Project 1, defining five different sized dots or blobs for stars of different brightnesses.
   (b) Define your own graphics blocks, to make better pictures as in Project 2.
   (c) Do the same to get more lifelike food in Program 7.
5. Try all these (and other) programs in different modes, so that you can build up your understanding of how the Electron has and uses these various screen layout systems. Try to become more familiar with the non-text modes (Ø, 1, 2, 4 and 5), with especial reference to their use of colour and numbers of lines and columns. The table that follows may help.

Table 5.1

Mode	Graphics	Text	Colours	Memory needed
Ø	256 × 64Ø	32 × 8Ø	2	2ØK
1	256 × 32Ø	32 × 4Ø	4	2ØK
2	256 × 16Ø	32 × 2Ø	16	2ØK
3	(text)	25 × 8Ø	2	16K
4	256 × 32Ø	32 × 25	2	1ØK
5	256 × 16Ø	32 × 2Ø	4	1ØK
6	(text)	25 × 4Ø	2	8K

The second column gives the 'graphics resolution' of each graphics mode; we'll refer to that later. The third column gives the number of lines and number of characters allowed

in a line. The fourth column gives the number of colours you can have on screen at once and the final one provides an indication how much memory the Electron will need to retain for itself if you are in that mode. Bear in mind that you can have only 32K units of memory when you switch the machine on, so that if 20K are used by the computer, that doesn't leave so much for yourself. Again this is a matter we shall return to later.

Note Please turn to Appendix 2 for some comments on these exercises.

6 Feast of Languages

As we've seen, everything a computer does is the result of instructions given to it by someone. All instructions must be in a form that the computer can act on. The instructions must be in the correct 'code' or 'language', if the micro is to 'understand'—in other words, the programming language BASIC in the case of the Electron.

Often the most efficient way of instructing a computer is to use a 'low-level programming language', such as its specific 'machine code' or 'assembler'. Efficient that may be, but easy it is not. In this book we shall not go as far as low-level programming—it is very interesting, and very rewarding, but best left to another time! Remember, though, that your Electron is already programmed in a low-level language—the permanent program in it that controls it and lets it understand BASIC can't be BASIC itself can it?

PROGRAMMING LANGUAGES

Until only a few decades ago, programmers had no choice but to work in low-level language. However, no doubt to their relief, the concept of the high-level language was then developed; BBC BASIC, as used by the Electron, is a high-level programming language.

Instructions given in a high-level language often look much like 'real' English. (Aren't English-speakers lucky?—there are very few high-level programming languages based on other human ones!) This means that they are easier for the programmer, but on the other hand, the computer must turn them into machine code before it can do anything with them. This translation takes time and memory so is not highly efficient.

There are hundreds of high-level languages. I am sure you have heard of some, like COBOL and FORTRAN. Apart from the BASIC we get with the Electron, a few other high-level languages are likely to appear for use with this micro in due course. That will need different chips inside, or as a half-way measure, machine code or BASIC instructions loaded from tape. Thus at the time of writing, when there were still far fewer than a hundred Electrons in Birmingham, one company had already produced a simple version of a particular programming system called LOGO Graphics (see Appendix 4).

The aims of each high-level language include ease of use, versatility, and independence. That last means that a program in the given language will be acceptable to any machine running the language. Unfortunately this doesn't happen as it should.

BASIC, in particular, being offered by more types of computer than any other programming language, suffers badly from the problem that most machines' versions ('dialects') differ from the rest. The manufacturer of each new micro almost perversely avoids standards in favour of new features. This gives progress, and it gives confusion and frustration!

BBC BASIC is more advanced than a number of other dialects, and that is good. But it does mean that BBC BASIC programs need some changes before other computers can run them, and vice versa. And, although the Electron's BASIC is close to that of its older

sibling, the BBC micro, the two systems are *not* identical and many BBC programs will need modifying before you can use them on your Electron. (There should be few problems in the other direction.)

As you are likely to want to try programs written for other computers on your Elk, here are some notes to help you. However, these notes cannot claim to be anything like comprehensive. Always refer to the *User Guide* for details, and even consider programming books for the other machine concerned.

COMPLETELY DIFFERENT

There is little, if any, relationship between the ways one can program certain things on different micros. The Electron's MODE, COLOUR, SOUND, graphics (drawing), random number system, and file-handling facilities are almost unique to it, even if they are close to those of the Beeb. Some computers indeed don't offer those features at all. The ones that do tend to use different keywords and different statement structures. Be careful too of differences in the numbers of characters in a line and the numbers of lines in a screen.

Use the programmable character system (VDU 23, Page 27) wherever a program calls for special keyboard symbols (as do the Sinclairs and PETs, for instance). Another area to watch out for concerns the instructions used with peripherals—especially printer and cassette machine.

The use of machine code/assembler routines in BASIC programs is fairly common with many modern micros. Keywords to watch out for here are PEEK and POKE (the Electron uses '?' for both), USR and CALL—but what one peeks or pokes or accesses by USR or CALL will differ greatly from machine to machine. Avoid trying to translate programs for other machines into Electron BASIC if they include any of those special keywords.

OTHER DIFFERENCES

ELSE This is not offered on all micros; it is a valuable feature of the Electron that we shall shortly explore.

END Some computers must have END as the last line of a BASIC program. This is not usually needed with the Electron—but it accepts it.

GET($)/INKEY($) There is much variation between machines in how to read the keyboard without using INPUT. Again we shall explore the Electron system in due course.

GOTO Some programmers use GOTO a lot, but the Electron BASIC doesn't much need it. As a program statement, GOTO 150 means skip over to line 150 and go on from there. As a command, GOTO 300 means start running the program from line 300. If you use this useful trick, remember that the line must exist—some computers, if told to go to a line that doesn't exist will start at the next higher line they find. With the Electron computer, by the way, you cannot use RUN followed by a line number—this is a pity!

LET Some computers must have LET in assignment statements: LET number = 4. The Electron does not, though, as I'm sure you have noticed, I always use it in this book. I do so because in my view assignment statements are then clearer.

REPEAT/UNTIL Like Electron ELSE, this is not available on most machines. Look out for the chance to use this structure to improve efficiency.

TAB The Electron's TAB (x, y) replaces the AT or @ of some other machines—but check the details of the numbers that follow it.

Arithmetic symbols The symbol for divide is / (on the bottom row), as in PRINT 12 / 4 giving 3, while * means times as in PRINT 12 * 4 giving 48. As far as I know, all versions of BASIC use those particular symbols. However, raising to a power, which posh folk call 'exponentiation', can use different symbols. $2 \times 2 \times 2$ (two times two times two) may also be thought of as 2^3 (two to the power of three, or two cubed). The Electron structure for raising a number to a power uses the symbol ^ (on the key with the left-arrow). Thus PRINT 12 ^ 3 gives 1728. Some computers use ** or ↑.

Functions (which we'll look at in a bit more detail later) are things like SIN and LOG; you may have met these in school maths. They need brackets in some BASICs. You *can* use brackets with the Electron, but usually you don't have to except to make the meaning clear. Thus you can have PRINT SIN (3Ø) or PRINTSIN3Ø with or without those spaces.

Names Although they can be any length you like (up to about 23Ø anyway!), in Electron BASIC the names of variables can contain only letters, numbers, £ symbols and dashes. (The dash, the so-called 'underline' symbol, is on the down-arrow key.) Of course if the names are of string variables you must add a dollar sign ($) at the end. The first character of these variable names mustn't be a number.

Goodness knows why you can include '£' in your numeric and string variable names. However the facility means that you can label very special variables with an initial £ (and what better symbol for something special?).

```
LET £3 = 3
```

(Some people use the underline dash for this purpose but it doesn't show up so well. On the other hand, £ causes problems with some printers if that's relevant to you!)

The names of files (which is what we call programs and data saved on tape) are a bit less restricted. These can be any combination of up to 1Ø characters, except that the first one cannot be a space or speech mark.

Do not use keywords to start variable names (TO, LET, and so on). Thus the Electron will not accept a variable name called TOM. However, the machine does distinguish between names using large and small letters (upper and lower case). This means that you *can* have a variable called 'tom', and that is one very good reason why I put variable names in lower case. (Another is that they show up better that way.)

As I've said, the length of names of variables in Electron BASIC is pretty well unrestricted. (Some computers allow only single character names; many let you use only two letters in a variable name, and so on.) With the Electron you can, if you so wish, use very long variable names. For instance, the machine would accept LET supercalifragilisticexpialidocious = Ø. But, imagine the problems in typing! Every time you want to use PRINT supercalifragilisticexpialidocious, you are in danger of letting typing errors mess up your program. While that is an extreme example, I would think that it is rarely necessary to have variable names longer than about six characters. I agree that so far I have used longer ones in this book, and indeed I shall occasionally continue to do so, but let's be reasonable! You must attempt to achieve a balance between a long meaningful variable name and a short less meaningful one that is easier to type and doesn't make your program listings look untidy.

Procedures These are invaluable for structured programming (see, for instance, Chapter 14). Most micros do not offer them, even though they have something which is rather similar.

Spaces The Electron requires spaces in instructions only if the meaning is not otherwise clear. So we can have PRINTSIN3Ø, PRINT SIN3Ø, PRINTSIN 3Ø or PRINT SIN 3Ø —the computer works out all these, so you may use whichever suits you. What I try to do throughout this book is to put in plenty of spaces in order to make program instructions clear to you, dear reader. I would suggest you do the same in the early stages of your programming. Some computer BASICs are always very strict about where spaces must appear. Yet others put spaces in for you, but not necessarily where you want them to be.

Of course, the Electron will not accept extra spaces as in PR INTSIN3Ø—again this is the same as most micros, except those that have keywords entered with a single key-stroke where the situation doesn't arise. (The Electron has three ways of entering keywords. You can spell them out in full, you can enter them in many cases by using FUNC with a single stroke and you can abbreviate them, as with using P. for PRINT. This versatility can lead to dangers . . .)

String handling is clumsy in BBC/Electron BASIC in comparison with what some micros offer. Be prepared for problems when converting Sinclair listings in particular. (Sinclair string-handling is a delight.)

Note There are many other differences between BBC/Electron BASIC and the dialects of other machines. However, what I've jotted down above covers the most important things, and will give you a good start to converting published programs into your dialect.

EDITING

This is as good a place as any to set out briefly the Electron's editing system. It is of particular value when you want to correct an error in one long line of a program, but has other time-saving advantages too. Say you have in your program '1ØØPRONT SIN 3Ø'.

1. That line may not be needed at all. Get rid of it simply by typing 1ØØ(R).
2. It may be totally wrong. Re-type the new version: 1ØØ INPUT answer. This is sensible too if the line is so short that method (3) isn't worth the trouble.
3. This is for use in making small changes in long lines.
   (a) Use LIST 1ØØ(R) to get the afflicted line alone at the foot of the screen. Below it will appear our two cursors: > and flashy _.
   (b) Press the up-arrow key. The flashy _ moves up leaving a blob behind it. Get the flashy cursor alongside the 1ØØ.
   (c) Press the COPY key (on the right-hand side of the keyboard). As you press it, the flashy cursor moves along the line 1ØØ and a copy appears next to >. Do this until the flashy cursor sits under the N of PRONT.
   (d) Now use the DELETE key to get rid of the O, and type the I you want.
   (e) Use the COPY key to finish off the line, and then press RETURN. *Don't forget this bit!*

That's editing! Much easier than correcting errors in a book!

Once you get the hang of the four arrow keys and COPY, you'll find it easy to construct lines out of all sorts of bits and bobs if you want. You can also edit the line numbers, so it is easy to produce loads of very similar lines from one. (I did quite a lot of this sort of thing when developing Program 9, for instance.)

On this subject, let me set out the use of LIST (FUNC + L, or L.).

1. LIST n puts line n over the cursor (if the line exists).
2. LIST, n gives all lines up to and including n.
3. LIST n, gives the program from line n onwards.
4. LIST n, m arrays before you the program lines from n to m inclusive.

Remember CTRL + N will let a long listing be 'paged', with SHIFT leading to the next page, ESCAPE letting you out, and CTRL + O reverting to the normal speedy scroll.

7 Join the Colours

Getting colour into your Electron programs is a complex affair, but one you just *have* to come to grips with. You may know that some other computers have colour keys—just pressing the key gives you the colour you want. The Electron is not programmed like that, but *does* offer you a much more flexible colour control system—once you've got used to it! The program fragment that follows is like the way we have begun many programs already. Let's start from scratch.

1. BREAK (and NEW if you haven't just switched on).
2. *KEY 1Ø"OLD|MLIST|M"—BREAK key gives BREAK, OLD and LIST.
3. The fragment—after you enter *each* line, use RUN to check the effect and BREAK to get back to the list.

```
1Ø MODE 5                    [MODE is FUNC + M, or MO.]
2Ø COLOUR 129                [COLOUR is FUNC + C, or C.]
3Ø COLOUR 2
4Ø CLS                       [no short way, but who needs it?]
```

And now some points. The effects of the colour commands depend on the MODE you are in. So if line 1Ø were not there, 2Ø and 3Ø would have no effect. (The Elk sits in MODE 6 unless you say otherwise, and MODE 6 isn't a colour mode.) Also lines 2Ø and 3Ø do not really show their effect until we do something else, such as instruct the screen to clear (line 4Ø, CLS).

So what does COLOUR do? Well, what it does is define the colours used when printing text. The paper (background) colour is given by a number greater than 127; the ink (text) colour depends on a number less than that. Table 7.1 shows the numbers and colours we can get in MODE 5 (and MODE 1) using COLOUR. The daggers (†) show what you get if you don't use colour instructions.

Table 7.1

Colour	Background	Text (foreground)
Black	COLOUR 128†	COLOUR Ø
Red	COLOUR 129	COLOUR 1
Yellow	COLOUR 13Ø	COLOUR 2
White	COLOUR 131	COLOUR 3†

I'm going to take a break from colour for a minute, to tell you about FOR . . . NEXT loops. That's because I want to use them in colour demonstration routines. Put your four lines on to key 1 for later use:

```
*KEY 1 "MODE 5|M COLOUR 129 |M COLOUR 2|M CLS |M"
```

Now, every time you press FUNC + 1 you get that set up. BREAK, and try it. Neat, eh?

FOR . . . NEXT LOOPS

As we saw when we met REPEAT (Page 13), a *loop* is a set of program lines that the micro cycles through a number of times. We use REPEAT . . . UNTIL for a loop when we don't know how many times to go round. (I also often use it as the last line in a program, to prevent the prompt re-appearing and to disable the keyboard—like this 99Ø REPEAT: UNTIL FALSE. See Program 6 for instance. You use ESCAPE to escape from this.)

When we *do* know how many times to loop we can use the FOR . . . NEXT structure. I used it once or twice already, to set up a certain delay, before the computer went on to something else. Here is *that* structure (this one from Program 9):

```
FOR wait = 1 TO 2ØØØ: NEXT wait
```

This gives a delay of roughly a second and a half, during which time the computer goes round the (empty) loop a couple of thousand times. In a second and a half!

Here's a FOR . . . NEXT loop that's a bit more meaningful. Enter it (after BREAK/NEW), so we can explore it.

```
7Ø  FOR number = 1 TO 1ØØ        [FOR is FUNC + F or F.; TO has no
                                  short form]
8Ø  PRINT; number "□";           [PRINT is FUNC + P or P.]
9Ø  NEXT number                  [NEXT is FUNC + N or N.]
```

(Both the semi-colons in line 8Ø are needed for my purpose, but try, and think about, the effect of leaving out each in turn if you have time.)

RUN, and try to explain what is happening; Figure 7.1 should help. What "FOR a = b TO c . . . NEXT a" does is as follows:

1. Set up the variable 'a', giving it the starting value 'b' (which must be defined already).
2. Carry out the statements inside the loop.
3. On finding NEXT a, return to the start of the loop and step the value of 'a' up by one. (Note that with the Electron, NEXT without the variable name is accepted, but I'll keep it in for clarity.)
4. Keep doing this until 'a' has passed the value of 'c', when the program leaves the loop.

Modify the above little program as follows, and try it out a few times.

```
1Ø  PRINT "First";
2Ø  INPUT first
3Ø  PRINT "Last";
4Ø  INPUT last
7Ø  FOR number = first TO last
8Ø  PRINT; number "□";
9Ø  NEXT number
1ØØ PRINT number
```

Are you happy with how it works? (Refer again to Figure 7.1 if you wish.) You should at least notice that the real true last value of 'number' is greater than you might think. (The effect of line 100 is to show this.) That means you must take care if you use the loop variable again after the loop is finished. But I hope you found no other problems. Did you try decimal numbers, zero, negative numbers among your inputs? Did you try a value of 'last' below the value of 'first'?

Most BASICs, including the Electron's dialect, extend FOR . . . TO . . . NEXT with the possibility of different step sizes. That is, the loop variable ('number' in our example) can change by jumps other than 1. Surprise, surprise—the keyword is STEP. Modify the program again to show this; it is now the final version, so why not sketch an up-dated diagram of it on the lines of Figure 7.1?

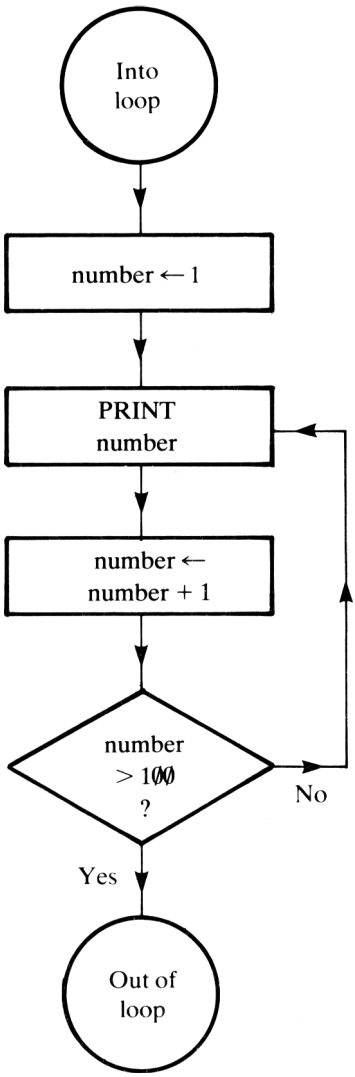


Figure 7.1

Program 10: The table of cubes

```
 10 PRINT "First";
 20 INPUT first
 30 PRINT "Last";
 40 INPUT last
 50 PRINT "Step";
 60 INPUT step
 70 CLS
 80 PRINT TAB(4) "Number" TAB(16) "Cube"
 90 PRINT TAB(4) "======" TAB(16) "====="
 91 REM  **  underline method  **
100 FOR number = first TO last STEP step
110   PRINT number, number^3
111   REM  **  ^ means 'raise to power'
120   NEXT number
```

Play around with that as you will. You may also like to compare it to Program 4 on Page 21. (You will find a BASIC arithmetic problem, no doubt—but we'll deal with that later.) But I wanted to do FOR . . . NEXT loops now so that we could investigate colour a bit more deeply. Remember?

CHANGING COLOUR

Use BREAK/NEW to clear out any existing program material. We still have our colour introduction stored on key 1, unless you've switched off since we got it there. Here's how to get it into our next program. The technique uses the nice AUTO facility. We met it before—it automatically feeds you line numbers from 10 up in steps of 10 (unless you tell it otherwise, and until you ESCAPE).

So—enter AUTO (R), and 10 appears. Now press FUNC + 1, and there are the first four lines of our colour program! ESCAPE to allow you to use your own line numbers and enter this *procedure*—'colourtest'. (Yes, I'll come back to procedures more fully later.) Read, but don't feel you have to copy, the REMarks.

```
 990 REPEAT: UNTIL FALSE
 991 REM  **  stops main program  **
1000 DEF PROCcolourtest
1001 REM  ** No space after PROC  **
1002 REM  **  PROC is FUNC+X  **
1010 CLS
1020 FOR star = 1 TO 640
1030   PRINT "*";
1031   REM  **  semi-colon needed  **
1040   NEXT star
1050 FOR wait = 1 TO 1000: NEXT wait
1051 REM  **  delay loop  **
1060 ENDPROC
```

The above lines, apart from 990, define the procedure I've called "colourtest". (990 simply separates the procedure from the main program.) The value of a procedure is that we can call for it to be used whenever we want. And we can develop it separately from the main program. Here's how we call it:

```
 10 MODE5
 20 COLOUR129
 30 COLOUR2
 40 CLS
 50 PROCcolourtest
```

RUN—to get a red screen full of yellow stars. The keyboard is disabled by line 990; use BREAK to get back to the listing. Now we can start colour changes! Add:

```
60 COLOUR 128
70 COLOUR 3
80 PROCcolourtest
```

OK? This shows we can use COLOUR when we like in a program, to change text and/or background colour.

The main program's a bit clumsy though—sure, you can change each colour statement in turn to investigate the effect. But let's try looping through all the possibilities . . . We're going to need new lines from 20 to 80, and a few other changes.

```
 10 MODE5
 20 FOR paper = 128 TO 131
 21   REM  **  'paper' is background  **
 30   FOR ink = 0 TO 3
 31     REM  **  'ink' is foreground  **
 40     IF paper = ink+128 THEN GOTO 80
 41     REM  ** Why?  **
 50     COLOUR paper
 60     COLOUR ink
 70     PROCcolourtest
 80     NEXT ink
 90   NEXT paper
990 REPEAT: UNTIL FALSE
991 REM  **  stops main program  **
1000 DEF PROCcolourtest
1001 REM  ** No space after PROC  **
1002 REM  **  PROC is FUNC+X  **
1010 CLS
1020 FOR star = 1 TO 640
1030   PRINT "*";
1031   REM  **  semi-colon needed  **
1040     NEXT star
1045  PRINT TAB(0,10) " Paper "; paper "  "' " Ink      "; i
nk "  "
1050 FOR wait = 1 TO 1000: NEXT wait
1051 REM  **  delay loop  **
1060 ENDPROC
```

Line 1045 tells the computer to display the names of the colours being used each time. If you want more time to read the message, increase the last value in the delay loop of line 1050. This program shows how we can use the four different colours for text and background that MODE 5 allows.

However, the Electron has on offer sixteen colours (or better, colour 'arrangements'). In MODE 2 one can have all sixteen on screen at once. However in MODE 5 (and also in MODE 1), you can select any four for showing in a given program. This is done by using another VDU statement to re-define the colours you normally get (the ones listed in Table 7.1).

Look again at that list of 'normal' colours. You can see the effects of the corresponding COLOUR statements using FUNC + 1, then COLOUR 0(R), COLOUR 3(R), and so on—four colours (including black and white) can be on screen at once. Having entered MODE 5 or MODE 1, we can use VDU 19 . . . to re-define any of those four. Thus VDU 19 128 4;0; (note the strange punctuation!—you can use commas rather than the second and third spaces) will make background number 128 blue instead of black. Then each time you use COLOUR 128, you get blue 'paper'.

Similarly if you would like to define COLOUR 2 (usually yellow ink) to be flashing yellow/blue (code 11), use VDU 19, 2, 11; 0;. Table 7.2 lists the codes for the sixteen 'colours' you can get with the Electron.

Table 7.2

Colour option	VDU 19 code	Flashing option	VDU 19 code
Black	Ø	Black/white	8
Red	1	Red/cyan	9
Green	2	Green/magenta	1Ø
Yellow	3	Yellow/blue	11
Blue	4	Blue/yellow	12
Magenta	5	Magenta/green	13
Blue-green (cyan)	6	Cyan/red	14
White	7	White/black	15

Using VDU 19 . . . gives a remarkable range of effects. It's tough going to get on top of this facility, but I'll now add to our existing colour program a routine to demonstrate it.

Program 11: Rainbow stars

```
 10 MODE 5
 20 FOR paper = 128 TO 131
 30   FOR ink = Ø TO 3
 40     IF paper = ink+128 THEN GOTO 8Ø
 41     REM  **  Skip 'invisible' combos  **
 50     COLOUR paper
 60     COLOUR ink
 70     PROCcolourtest
 80     NEXT ink
 90   NEXT paper
 99 REM  **  Start new routine  **
100 COLOUR 128
110 COLOUR 1
120 FOR paper = 1 TO 15
130   FOR ink = 1 TO 15
140     IF ink = paper THEN GOTO 160
150     PROCcolourtest
160     VDU 19, 1, ink;Ø;
170     NEXT ink
180   VDU 19, 128, paper;Ø;
190   NEXT paper
990 REPEAT: UNTIL FALSE
1000 DEF PROCcolourtest
1010 CLS
1020 FOR star = 1 TO 640
1030   PRINT "*";
1040   NEXT star
1045 PRINT TAB(Ø,1Ø) " Paper "; paper "  "' " Ink      "; in
k "  "
1050 FOR wait = 1 TO 1ØØØ: NEXT wait
1060 ENDPROC
```

If you've got the patience, enter different modes by direct command (such as MODE 6), and then GOTO 1ØØ. You see that the ink and paper colours in even the '2-colour' modes (Ø, 3, 4, and 6) can be re-defined using VDU 19 . . .

As I say, it is up to you whether to go further or not at this stage. It *is* important for a number of programming needs, so you may like to use the next little program as a basis for study and practice.

```
10 MODE 5
20 VDU 19, 128, 4;Ø;
21 REM  **  Gives blue paper for black
30 COLOUR 129
```

```
 31 REM  **  Prepares for red paper
 40 CLS
 41 REM  **  Clears to RED paper
 50 COLOUR 128
 51 REM  **  Prepares for blue paper
 60 PRINT' "Hallo!"''
 61 REM  ** Uses BLUE paper
 70 COLOUR 3
 71 REM  **  Prepares for white ink
 80 COLOUR 2
 81 REM  **  Prepares for ink colour 2
 90 PRINT "And again!!!"''
 91 REM  **  Uses white/black ink
100 VDU 19, 2, 11;0;
101 REM  **  Sets ink 2 to be y/b
110 VDU 19, 2, 15;0;
111 REM  **  Sets ink 2 to be w/b
120 COLOUR3
121 REM  ** Prepares for ink 3 (white)
```

There is only one crucial thing to note—from your observation of the effect of lines 100/110 on line 90. It is that you can put the VDU 19 . . . statements anywhere in your program, but if you have more than one for a given standard colour, only the last will have any influence. I shall have to take another look at colour programming when we get to Electron graphics, so you *can* skip it for now—up to you. Here, meanwhile, are some projects on what we've done so far.

PROJECTS

1. Write a MODE 5 program to flash the screen red-yellow-red a hundred times. If it doesn't work first time (and I bet it doesn't), try slowing it down!
2. Rewrite Program 2 or 4 and Program 5 using FOR . . . NEXT instead of REPEAT . . . UNTIL. Try to understand the situations in which each loop structure is the better.
3. Experiment with:
   (a) changing colour in mid-screen;
   (b) the flashing facility.
   (Experimentation is the best way to get on top of VDU 19!)
4. Write a program offering the user choice of colour from the main six, giving him/her a screen of that colour, and then . . . I don't know, *you* think of something.

8 Command Performance

In this chapter I'd like to draw your attention to, and extend my coverage of, rather a ragbag of Electron facilities that will make your programming faster and more effective. First, let me use the opportunity to remind you of the main commands we've met. (A command is an instruction for the computer that's entered directly, without a line number, and carried out at once. Recall that many instructions normally used as program statements can in fact also be entered as commands.)

COMMANDS

We have so far met several commands, four being of use when there's a program in memory. These are OLD, LIST, RUN and NEW. Each needs to be typed out and followed by the return (R) key before being carried out (executed). And we've met AUTO, mainly for adding a program to memory, and SAVE, CHAIN and LOAD, which apply to tape (or disc) transfers. Here are fuller details, including practical notes, on all the Electron's commands.

AUTO puts up line numbers on screen for you to add the intellectual parts to. It goes on until you press ESCAPE. AUTO alone gives numbers in steps of 10 from 10 onwards. However, you can choose other start and step values if you like:

AUTO start, step

Wherever you start, the step value remains at 10 unless you say otherwise—but don't, whatever you do, try a step value greater than 255. Well, try it just this once—and read Acorn's most amusing (supercilious?) error report!

CHAIN see Chapter 10.

DELETE see Chapter 18.

LIST displays on screen, in line number order, the current program instructions. This command has several forms, as I started saying a few pages back.

LIST displays the whole program.
LIST n displays only line number n (if it exists).
LIST n, shows the whole program from line n onwards.
LIST, n shows the whole program up to line n.
LIST n, m displays the section from line n to line m.

When you have a program of more than a couple of dozen lines LIST by itself becomes rather cumbersome. It's not easy to find the bit you need to look at, nor is it easy to edit.

If the program has more than 25 lines it can't all fit on to the screen at once in MODE 6. The list then 'scrolls' through until the last line is displayed. LIST such and such gets over that. You may prefer instead to stop the scrolling action. That is done by entering CTRL + N (no (R) needed). CTRL is the 'control' key (marked CTRL, over on the left of the keyboard); to use it, hold it down while pressing the N key. The Electron will no longer scroll its program lines past you at amazing speed, but stops at the end of each few lines. In this so-called page mode, you need to press the SHIFT key to move on. You can return to normal speedy scrolling by using CTRL + O or BREAK.

Another LIST option is called, surprisingly, 'list option', the keyword being LISTO (*not* LISTØ). Using this will get the machine to force spaces into your listings, to make them more readable (but take up more memory). The number you use with LISTO fixes exactly how this is done. Here are the options.

LISTO Ø: no spaces (normal)	LISTO 4: indentation in REPEAT loops
LISTO 1: spaces after line numbers	LISTO 5 = LISTO 1 + LISTO 4
LISTO 2: indentation in FOR loops	LISTO 6 = LISTO 2 + LISTO 4
LISTO 3 = LISTO 1 + LISTO 2	LISTO 7 = LISTO 1 + LISTO 2 + LISTO 4

After LISTO n, whenever you list, you'll get your option of spaces. So if you want all possible spaces and indentations, enter LISTO 7 (R). Whenever you LIST thereafter, you will find the posh program layout that some books and magazines use for maximum clarity. Including this one of course. (There is a problem with LISTO, though; when you're editing you'll find it can force in extra spaces on edited lines that can make the result very untidy. Because of this I don't use LISTO until I've given my program its final polish.)

LOAD see Chapter 10.

NEW is not new to us (ho, ho). Use it to remove the current program from memory and put all variable values to zero. NEW does not de-program the functions on the number keys or the CONTROL instructions you've previously entered.

Some folk don't bother with NEW, but avoid their error! If the new program doesn't use exactly the same line numbers as the old one, you get a crazy, mixed-up program—a combo of instructions from two programs in one. I guess that won't work the way you want! Also *don't* get into the habit of switching the micro off to clear the memory. Well, the practice shouldn't do any harm if you have a standard model Electron, but it could cause damage to equipment you may later add on—and by then it'll be too late to change your habits.

OLD logically as well as alphabetically follows from the previous entry. Used after BREAK, it lets you access again the program that was in memory. If you use BREAK and then enter a number, followed by any character and (R), the computer 'assumes' that you've put in a line number of a new program. It then in effect clears the old lines from its memory. So always use OLD after BREAK, if you want to keep that program around. Indeed, as you know, I recommend that you program the BREAK key specially to give OLD on its own. If you want to get rid of a program, use NEW (R), 1 □ (R) and not even BREAK, however programmed, will get the old stuff back for you.

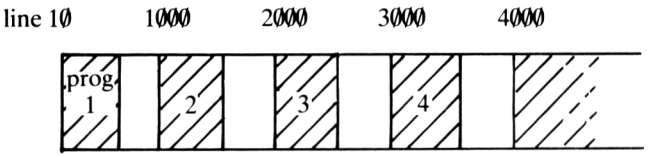
RENUMBER see Chapter 18.

RUN should present no problems by now. It tells the computer to start carrying out the instructions of the current program from the line with the lowest number. It also sets all your variable values to zero; if that is not what you want, command GOTO n (where n is the first line number) instead. The command GOTO such and such is also useful in other cases:

1. When you want to start the program running from a different point than the start, to test how one part works, for instance.

2. When you have several separate programs in memory, and want to execute any but the first.

Figure 8.1 should make this clearer.



Get Program 1 with GOTO 1Ø (or RUN)
Program 2 with GOTO 1ØØØ
etc.

Figure 8.1

Some micros let you use RUN n in such cases; sadly the Electron does not.

SAVE see Chapter 10.

EDITING

By now, a swish programmer like you will need to know even more about editing than we've done so far, so check through this bit, best practising on a program in memory to get used to the techniques. The full set of editing facilities to change program lines on the Electron is very versatile. That makes it a bit clumsy if you're not careful.

The simplest form of editing concerns the complete removal of one or more lines of the program. That's easy! You just enter each line number in question followed by (R). The line has now nothing in it, so is nothing—and it vanishes from the listing. But *do* be sure you want to remove the line before doing this! If you are *not* sure and you want to see how a program works without a certain line, insert REM after the line number. REMember that REM keeps the following material in the listing, but the computer ignores it. If you're not sure how to insert REM into a line, just read on.

The same kind of technique allows you to replace an existing line with a new version. Thus if line 5Ø is PRINT an$ and you want it to be INPUT an$, just enter 5Ø INPUT an$ (R)—and this will now appear in the list instead of the old form.

Fine so far—but it's a bit of a drag to go through all that if the old and new lines are lengthy, or if minor changes are needed to many lines. Then we can call on the Elk's *full* range of editing facilities. These use the four *cursor control* keys—the set of four keys at top right with arrows in different directions, up, down, left and right, on them—plus the COPY key beside them.

Press the COPY key now. It will beep at you, sadly pointing out that there is nothing for it to copy. So let's give it something to copy. Enter a couple of fairly complex program lines. Something like these:

```
1Ø PRINT "This line is wrong."
2Ø PRINT "This line needs copying."
```

How can we correct line 1Ø to make it right? We may want it to read 1Ø PRINT' "Now this line is right." We could do it by just entering the new version, but let's try screen editing. Press the up-cursor key twice. The dash prompt moves up screen to lie next to the 1 of line 1Ø and leaves a blob behind. Press COPY. No beep now—the result is that the first character or two of line 1Ø appear—are copied—as the blob moves along. Follow through this sequence of key presses; I'm starting again.

Action	Result by > prompt	Effect on _ prompt
1. ↑↑	Blob appears: > ■	Up two lines
2. COPYYYYYYYYY (repeated)	> 1Ø PRINT ■	Across ten places: "_
3. Apostrophe	> 1Ø PRINT ' ■	None
4. COPYY	> 1Ø PRINT ' "T ■	Across two places: h
5. DELETE	> 1Ø PRINT ' " ■	None
6. Now t	> 1Ø PRINT ' "Now t	None
7. COPYYYYYYYYYYYY	> 1Ø PRINT ' "Now this line is ■	Across twelve places: w
8. right."	> 1Ø PRINT ' "Now this line is right." ■	None
9. (R)	> _	Returns next to >

Complicated? Well, maybe. Easier to do than describe. Let me summarize. The cursor control keys can move the _ cursor anywhere on the screen. When you press COPY, by the > prompt appears the characters underlined by the _ cursor. This goes on until you stop pressing COPY. At any stage you can delete one or more characters (with the DELETE key) or insert one or more by typing them as usual. When you press RETURN, the end of the line is assumed and we return to the usual position.

Try to use these keys—cursor control, COPY, DELETE and RETURN—to get the following versions of line 1Ø.

```
1Ø  PRINT "This line is very wrong, this line is."
1Ø  PRINT "This line is wrong, wrong, wrong."
```

We can edit line numbers as well, to give new ones. Work out how to use edit to repeat line 2Ø in lines 3Ø, 4Ø and 5Ø. Finally, edit line 1Ø, to read:

```
1Ø  PRINT ' "This line is wrong, wrong, wrong—it needs copying."
```

When you edit in practice, take note of this final tip. Use LIST n, m to get just the few lines you want to work with. Otherwise the process takes a lot longer and is more open to mistakes. Mistakes? Well, there is one very easy trap to fall into. While editing a line, do *not* press RETURN after insertions/deletions, but COPY the rest of the line first. Otherwise you'll end up with only half the line you want. Remember; the micro assumes you're at the end of the line when you press (R). Even if you're not.

COMPUTER MEMORY JARGON

The computer must be able to *store* the program instructions and data it is currently working with. It must also be able to store the *operating system* instructions—all the details of how it should carry out the tasks it can be given.

All these things are *data* as far as the computer is concerned—electronic representations of things called binary numbers. The computer's circuits handle strings of binary numbers following certain rules. The binary number usually has the name *bit* (= binary digit); it can take the value of Ø or 1.

A *byte* is a set of eight bits. The value of a byte can range from Ø (ØØØØ ØØØØ) to 255 (1111 1111). In the case of most modern micros, including the Electron, the byte is the same as a *word*, a set of bits of standard length. (You may also come across the charming *nibble*, a set of four bits.) The maximum size of a micro's memory is usually 65536 bytes. Computer folk call this 64K bytes or 64K for short. Here K stands for Kilo; *1 Kilobyte* is 1Ø24 bytes. This is close to 1ØØØ, called kilo; the use of the capital letter is to remind us that the figure is a bit bigger than 1ØØØ. We measure the actual size of the computer

memory in Kbytes as well. And the unit has all kinds of other related uses.

The main memory of the micro is divided into two parts. The operating systems and so on are permanently fixed in *ROM*. This is 'read only memory'—data stored so that it can be got at (read) but not changed. The Electron ROM is 32K, meaning that it contains 32768 bytes of operating instructions which can't be altered. The contents of ROM are truly fixed, even when the power is off.

With 32K of the Electron's 64K maximum taken up by ROM no more than 32K is left for the user's programs and data. We must be able to change these, so ROM is no good—we have *RAM* for that. RAM is 'read and write memory'—the contents can be got at (read) and also changed (written to). (Some folk think that RAM is 'random access memory', but that is not so. Both RAM and ROM are random access memory—we can reach any part of them directly without starting from the beginning.)

It is important to note that RAM size never tells you how much memory the user can actually employ—the computer itself needs quite a few K of RAM for its own seedy purposes.

Very crudely, each character you tell the computer to remember takes up one byte of memory. Each byte of memory taken up is one less available for later use, and one more to take time during saving and loading operations. The 32K offered by the Electron (in practice far less than that) is not really a very large amount, so that in this chapter and elsewhere I do think about saving memory. All those spaces I put in program listings for clarity are really a burden; so too are REMs and single-statement lines. It is, however, important to balance clarity with memory miserliness.

How then can one tell how much memory is left when part way through developing a program? Well, I have mentioned things called PEEK and POKE before, and you need to venture a little way into that area for this purpose. If you enter the instruction that follows on one of your number FUNC keys, you can access it whenever required to find out how much memory you have left free.

```
*KEY n "PRINT (?5 - ?3) *256 + ?4 - ?2 ¦ M"
```

It is an eye-opener to use this when you first switch your Electron on—you get an answer far less than the 32K bytes you might expect. Remember those seedy purposes?

SQUASHING PROGRAMS

The Electron offers two main facilities for reducing programming time and memory. The first is that keywords can be abbreviated, or in some cases, got with the use of FUNC; the second is that more than one statement can be placed on a single line. The first saves programming time and reduces typing errors; the second reduces memory take-up and *can* make listings easier to follow.

Abbreviated keywords are acceptable to the micro when you are entering commands or instructions. In each case, enter one or two (or sometimes even more) letters plus a full stop, instead of the full word. Thus you can use P. for PRINT, L. for LIST, and AU. for AUTO. When the abbreviated instructions are listed, they appear in the usual form, and take up the usual amount of memory.

This is obviously a useful technique for the longer common keywords. However it's not worth the trouble of thinking about it in less common cases. Thinking *is* necessary, as, with well over a hundred keywords to play with, the abbreviation used cannot be just the first letter. Here is a list of the keywords met so far, and the abbreviated forms.

I think that those marked with a dagger (†) are hardly worth the effort as they save so little time. You may think that that is true about some of the others as well. The star (*) we'll come to in a moment.

To confuse the matter further (but to be complete) note also that you can use longer abbreviations than those shown if you want—IN. or INP. (or even INPU.) for INPUT is an example.

AUTO	AU.*	MODE	MO.†*
CHAIN	CH.*	NEXT	N.*
CHR$	CHR.†	OLD	O.*
COLOUR	C.*	PRINT	P.*
DELETE	DEL.	RENUMBER	REN.*
ELSE	EL.†*	REPEAT	REP.*
ENDPROC	E.	RUN	RU.†*
FALSE	FA.	SAVE	SA.†*
FOR	F.	STEP	S.*
GOTO	G.*	THEN	TH.†*
INPUT	I.*	UNTIL	U.*
LIST	L.*	VDU	V.*
LOAD	LO.*		

The third way of getting a keyword into the Electron's tiny mind is to use the FUNC + letter key system, as you know. This is fairly fiddly, but well worth the trouble of learning. Hold FUNC down and press a letter key, and whatever is etched on the front of the key will appear on screen—in statements, inputs, strings, any old time. Try making up messages out of FUNC words if you have time to waste.

All the keywords marked * in the table above are FUNC ones. Note, by the way, that OLD and RUN when got by FUNC have an (R) thrown in for free. Here are some further points.

One or two etchings on the key fronts aren't the same as you get on screen. Also it's too much trouble learning all those key codes—concentrate on the common ones. In the above list I have starred those you may have come across by now, but I don't think they're all worth learning. Best is to go through the list in the *User Guide* (Page 42) and cross out the ones that you don't think you'll ever need to bother with. The final point about FUNC is that the method does not lead to any memory saving.

Multiple statements on a line is the posh term for structures like the following; we've already used one very like this in fact:

```
100  FOR wait = 1 TO 10000: NEXT wait: REM delay
```

This contains three statements (FOR . . ., NEXT . . ., and REM . . .). There's nothing wrong with putting them on three lines, but they *can* go on one with colons to show the boundaries between them.

It is logical to use this technique in cases where a small number of lines have one specific task, as with the delay loop above. Do *not* overdo the approach, as it can make a program less easy to read. Also there are cases where you will introduce bugs if you try to do this too much. However you can have up to about 20 or 30 statements in one line like this if you wish (it depends on their length of course). Here are some more examples:

```
120  CLS: REPEAT: LET star = star + 1: PRINT "star";: UNTIL star = 640:
     REM star fill
130  REPEAT: CLS: PRINT TAB(0, 10) "Message";: INPUT answer:
     PROCtest: UNTIL fail = 0
```

The need to PRINT "message" close to INPUT such and such is so common that the Electron has a special squashing trick just for that. Here it is:

```
150  INPUT "Message", answer                    [gives ? prompt]
```

or alternatively

```
150  INPUT "Message" answer                     [no prompt produced]
```

Here's the structure:

line number INPUT "Message" (,) variable name.

The assigned variable can be a string or a number, of course. More than one input at a time can be accepted too. In other words, with the Electron, you can use instructions like this:

```
200  P. "Enter three numbers, with commas between, and RETURN."
210  I. a, b, c.
```

Both are in effect useful multiple-statements-on-a-line techniques. Try them in *your* programs; note them in mine. (And now recall the point I made a few pages back about commas in input strings—Page 21.)

COMMENTS

Let me close this chapter by noting a few other points of interest to programmers.

More on *FUNC*. Each of the number keys can be programmed like the BREAK key (which counts as number 10). So you can do this:

```
*KEY 3 "FOR wait = 1 TO 1000|M NEXT wait|M"(R)
```

Here

*KEY n (or *key n) is the instruction to program key number n.
". . ."—the instructions appear in a pair of quote marks (though in fact the quote marks are optional)
|M is the code for RETURN in this context.

As with multiple statements on a line, you can have one or more instructions between the quote mark and the unquote mark. It is simpler to separate them with colons rather than with |M—but you must close with |M if you want to avoid using RETURN after using the programmed key. So the above *could* be written like this:

```
*key3FORwait=1TO1000:NEXT:REMdelay|M
```

Now there's compression for you!

The only advantage of using the "|M" thing rather than the colon is if you want to use a set of instructions with AUTO as I demonstrated on Page 40. Also some keywords, OLD and LISTOn included, *must* have after them (R) or |M, as the case may be. Note that you can use the *KEY as part of a program if you want, though that implies the user of the program knows how to use FUNC.

The DELETE key Each time you press this, the last character in the material being entered or edited is deleted. Keep the key down to delete a lot of characters. Users of your programs can use DELETE to correct an input answer before pressing RETURN. I wish I had a key like this on my typewriter; it's one of the nice things about text editing (word processing)—so easy to wipe out errors!

REPEAT If you hold any key down for more than about half a second, you will find its effect repeated as long as the key is down. This is particularly useful with cursor control, COPY and DELETE, but works on *all* keys. Underlining and printing patterns become very easy.

9 Getting Your Sums Right

If you think about it for a moment, you'll realise that the word 'computer' implies something for doing mathematical calculations. The first machines of this kind were certainly just programmable electronic calculators. (And they weren't as good at calculating, in many ways, as the pocket calculators we can now buy for a few pounds.)

When I defined the computer near the beginning of this book, I spoke of it as a data processor. In fact whatever kind of data a computer processes, it all has to be represented in number form (binary numbers in fact) and all the processing consists of mathematical processes. OK—that's not very relevant to us as users. But it *is* time to find out a bit more about arithmetic in Electron BASIC. We've met it quite a lot already, and I hope it didn't worry you. Anyway, let's look at this little program. Enter, use, consider.

Program 12: Four-function calculator

```
   10 MODE 5: COLOUR 130: COLOUR 0
   11 REM  **  Arithmetic CAN be colourful!
   20 REPEAT
   30   CLS
   40   PRINT TAB(20) "Enter two numbers:"'"RETURN after eac
h!"''
   41   REM  **  Multiple input...
   50   INPUT num1, num2
   60   CLS: PRINT num1, num2
   70   PRINT'' "Sum", num1+num2
   80   PRINT'' "Difference", num1-num2
   90   PRINT'' "Product", num1*num2
  100   PRINT'' "Quotient "; num1/num2
  101   REM  ** Care with layout here!
  110   FOR wait = 1 TO 1000: NEXT wait
  120   PRINT TAB(0,25) "Press RETURN for"'"the next go...."
''
  130   INPUT nextgo
  140   UNTIL FALSE
```

Note the use of INPUT in line 13Ø—it holds the display until the RETURN key is used. The input value, if any, is ignored—this is simply a useful 'hold' trick. Note, too, the double input at line 5Ø. The message asks the user for two numbers separated by (R). In fact, you may note that this structure will also accept two numbers separated by a comma.

Anyway—the arithmetic. The above program shows simply how BASICs deal with the four main arithmetic functions: add, subtract, times and divide. Note the special symbols '*' for times and '/' for divide.

We have already met a fifth important function, given by ^. The posh name for this is—as you know from Chapter 6—'exponentiation'. I call it 'raise to a power', but that's a mouthful too. Add this to the above program:

```
95  PRINT'' "Power", num1 ^ num2
```

Again, try it.

I hope when you're running this program, you're testing it fully. What happens when you input decimal numbers, negative numbers, zero, and so on? What happens at line 1Ø if 'num2' is zero? (Computers can't divide by zero—why not?) And what would happen if we had more complex expressions to work out than those in Program 12? Think about this little problem . . .

Two people each have three ball-points and two felt-tips.
How many pens have they altogether?

OK. I know you can work it out in your head. But how do you do it? And, more important, how do you tell the computer to do it? Does *this* give the right answer (try it!)?—PRINT 2 * 3 + 2. I hope you tried it; I hope you got the 'wrong' answer. But computers can't make mistakes, can they? What's happened to the missing two pens? Try *this* then—PRINT 3 + 2 * 2. After all, that's how you do it in your head, and it's how you would do it on a hand calculator. Wrong answer still? A *different* wrong answer. So there's a problem. It's not the computer's fault. The thing is that the machine is trained always to multiply before it adds. So it goes

2 * 3 + 2 = 6 + 2 = 8 in the first case

and

3 + 2 * 2 = 3 + 4 = 7 in the second.

The posh term for this is: multiply has a higher priority than add. If you want to break priority, as we do here, you have to use brackets: (). (Note that the Electron has square and curly brackets, on the COPY key and its two neighbours, but they are for something else, and don't work here.) So, you must enter PRINT 2 * (3 + 2), and the computer goes 2 * (3 + 2) = 2 * 5 = 1Ø
IF YOU DON'T PUT BRACKETS WHERE THEY'RE NEEDED, YOU'LL GET THE WRONG ANSWER .

On the other hand, if you use them where they're *not* needed, you get the right answer. So—if in doubt, put them in. Here is the Electron's priority list as we need to know it so far.

Highest	(. . .)	brackets
↓	. . .	functions (like SIN)
	^	raise to power
	*, /	multiply, divide
Lowest	+, −	add, subtract

There are others in the full list, but we needn't worry about them yet. Now, if you're not mathematically minded, you're not likely to want much in the line of sophisticated number-crunching programming. You have my permission to skip the rest of this chapter, then—but please glance through the projects at the end. One day you'll want to come back, I hope . . . but first

An important note

After Program 1Ø, I noted that you are likely to have found a BASIC arithmetic problem. It is that sometimes computer arithmetic calculations do not give exactly the correct answer. While PRINT 2 * 3 gives 6, there may be certain occasions when you might find the answer 5.99999999 or 6.ØØØØØØ1.

The reason for such occasional problems is the way computers handle numbers. In fact they don't work with decimal numbers, but what we call binary ones. A simple number in decimal may become quite complex in binary form and when the computer tries to round it off it doesn't do so properly. It is rare to find arithmetic errors appearing in the first few decimal places of an answer, though even that can happen sometimes, so normally you would avoid using the last few significant figures.

Anyway, these things *do* happen, and it's not a fault in the computer. (I have to say

that because people often get worried about such apparently strange results, and write to the computer magazines or the manufacturers to complain, or even phone me up!)

A less important note

Just as well say this here. Don't get carried away with exponentiation, I mean with using ^ . It's quite slow in fact. PRINT x * x is a lot faster than PRINT x ^ 2. Squares, cubes and fourth powers are best found using * rather than ^ ; also PRINT 1/(x * x * x) is faster than PRINT x ^ −3. Save ^ for more unusual powers!

NUMBER-CRUNCHING

Right—if you're still here, let's go on. If you have a bit of a mathematical background, you may have come across things called quadratic equations. These are equations like $ax^2 + bx + c = 0$, where a, b, c are numbers (coefficients) and x is a variable. For any set of values of a, b and c, there can be no more than two values of x that fit the equation. We call these two values the 'roots' of x. At school we had to find them like this:

$$\text{Root 1: } x_1 = \frac{-b + \sqrt{(b^2 - 4ac)}}{2a} \qquad \text{Root 2: } x_2 = \frac{-b - \sqrt{(b^2 - 4ac)}}{2a}$$

Here is a program to find the roots of x for any input set of coefficients a, b and c, if the roots exist. (They don't always!)

Program 13: Quadroots (improvements follow)

```
  10 MODE 5: COLOUR 129: COLOUR 0: CLS
  20 PRINT TAB(21) "Enter coefficients"' TAB(1) "===== ====
========"
  21 REM  **  Underlining
  30 INPUT''' "a = ", a
  39 REM  **  Now use EDIT
  40 INPUT''' "b = ", b
  49 REM  **  And again
  50 INPUT''' "c = ", c
  60 LET x1 = (-b+SQR(b^2-4*a*c))/2*a
  69 REM  **  And again!
  70 LET x2 = (-b-SQR(b^2-4*a*c))/2*a
  80 PRINT''' "Roots:"'', x1'', x2
  90 FOR wait = 1 TO 2000: NEXT wait
 100 INPUT'''' "Again",again: RUN
```

The important lines to us here are 60 and 70. Study them in the light of what I have just said about priorities. And I *mean* study: for each line work out the order of computer operations and the need for each part. SQR is the square root function. We can use it like this—P.SQR 4, giving 2—in simple cases: otherwise put brackets round the numerical expression as I did in Program 13.

The square root of a negative number has no real meaning and cannot be evaluated by the micro. Thus you will get the *error message* "−ve root at line 60" if a = b = c = 1. And the program stops. To avoid this kind of problem the Electron has a useful *error-trap* facility. The statement structure is ON ERROR (do such and such). Try this—it's worth bearing in mind, but needs *great* care in use:

```
5 ON ERROR PRINT' '"No real root!"' ': RUN
```

Now we've 'trapped' the error—provided a more relevant message and avoided the program stopping. But the danger is it'll do the same whatever the error is. In particular if you press ESCAPE, previously the best way to get out of this program, you will find that you get the message "no real roots!", and the program keeps running. This is of

course a useful technique for preventing people from escaping your super programs—but it's not really satisfactory if you wish to allow them to use ESCAPE. At the moment I shan't say anything more about ON ERROR, but clearly we'll have to look at it in more detail later. When running the above program, therefore, the only way you can escape is to use BREAK.

Well, actually I ought to say a little bit more about *that*! If you have pre-programmed the BREAK key using the following structure:

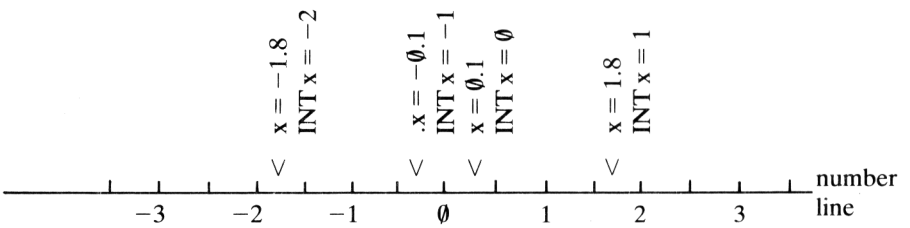
```
*KEY 10 "OLD|M RUN|M"
```

you won't be able to get out with that either! The only way to escape from a BASIC program which has trapped the ESCAPE and BREAK keys in this way, is to press CTRL + BREAK.

The next function I'd like to discuss is INT. Try the following direct commands to test what it does:

```
P.INT x,
```

where 'x' is each one of a set of numbers like 1.2, 1.25, 1.8, 0.1, and 0 and similar negative ones. I hope you can see that what INT x does is to return the whole number (integer) that comes to the *left* of 'x' on the number line. See the sketch, Figure 9.1. We can often use INT just like that. There are two particularly useful INT structures, however, that you should get to know. We'll use the second in a minute to make a further improvement to Program 13.



1. We can use INT to round a mixed ('decimal') number to the nearest whole number—to give 1 from 1.2, and 2 from 1.8, for instance. Here's the structure:

   ```
   LET x = INT(x + 0.5)
   ```

   Try it on a set of numbers as above; refer to the number line sketch if you can't see how it works.
2. We can also use INT to round a mixed number to a value correct to a given number of decimal places. This is particularly valuable to get round the problem of computer arithmetic mentioned a couple of pages back. To two decimal places, for instance (as when dealing with decimal money) we want 1.23 from anything between 1.225 and 1.234999 . . . Here's *this* structure:

   ```
   LET x = INT(x * 100 + 0.5) / 100
   ```

   Test it out fully as usual. Adapt it to give a different number of decimal places.

We can now go back to Program 13 and modify it to give roots correct to (say) four decimal places. Edit lines 60 and 70 to include INT like this:

```
60  LET x1 = INT( ( (-b + SQR(b ^ 2 - 4 * a * c) ) / 2 * a) * 100000 + 0.5) / 100000
```

OTHER FUNCTIONS

I'm just going to give you a bald list of the main mathematical functions concerning numbers offered by the Electron, with short comments where necessary. If you're going

to use any, I guess you'll have a fair idea of their meaning. Note I use the word 'argument' occasionally—it doesn't imply anything awful about my pacific nature, but is the official name of whatever the function acts on. If you ask me why, I can only quote the Oxford English Dictionary which mentions the Latin *argumentum*, meaning refashioning.

ABS returns the absolute value of the argument. Thus ABS 4 = 4; ABS −4 = 4. Its main simple use is to evaluate the absolute (actual) difference between two numbers:

```
IF ABS(guess − answer) < 10 THEN PRINT "Well done!"
ELSE PRINT "Wrong!"
```

ACS, *ASN*, *ATN* return the values of the angles whose cosine, sine and tangent (respectively) are the arguments. The angles are given in radians:

```
ASN 0.5 = 0.523 . . .
```

Use DEG to convert to degrees:

```
DEG(ASN 0.5) = 30
```

COS, *SIN*, *TAN* give the cosine, sine and tangent (respectively) of the arguments. The arguments are assumed to be in radians:

```
SIN 30 = −0.988 . . . (did you expect 0.5?)
```

Use RAD to convert to radians:

```
SIN(RAD 30) = 0.5
```

DEG converts radians to degrees (see ACS above).

DIV returns the whole number part of a division. (MOD gives the remainder.) The structure may look strange:

```
P.100 DIV 9 gives 11 (whole number)
P.100 MOD 9 gives 1 (remainder)
P.100 DIV 9, 100 MOD 9 gives both
```

LOG gives the logarithm of the argument to base 10.

MOD See DIV above.

RAD See COS above.

RND We've looked at RND already. This gives some kind of (semi-)random number in a range depending on its argument, if any. For instance:

```
RND(12) could give 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 or 12.
```

RND by itself gives a whole number at 'random' between −2 147 483 648, and the positive version of the same. It is quite often useful to use RND(1), giving a random number between 0 and 0.9999999. (That's what the *User Guide* says, but it's going to take rather too long to check!)

SGN is of occasional use. SGN x gives −1 if x is negative, 0 if x is zero, and +1 if it is positive. Thus:

```
IF SGN x < 0 THEN LET x = −x
```

SQR (dealt with earlier) gives the square root of its argument. Thus SQR x = x ^ 0.5. If the argument is negative an error results, but you found that out already from dealing with Program 13.

Finally, a not-function, but I need to put it somewhere. The Electron stores the value of π permanently. Get it with PRINT PI, and see the value 3.14159265. In fact, I think the value held in the machine is greater than that, but I'll leave it to you to investigate the matter if it's of interest.

DIY FUNCTIONS

As you have seen (if you're still reading this!) there are many useful functions in Electron BASIC (most of the above are provided with most micros of course). But, if you're not satisfied and have some other function you often want worked out—you can define it and then use it just like the others. I'm going to give only brief details of what to do here. Most readers will not have the need—if *you* do, read these notes and then if necessary turn to the *User Guide* (or to a fairly good BASIC textbook). There's an example in Program 18.

Step 1

Use DEF FN (define function) to tell the micro what you want. Say you want a function called power whose value is given by $x^3 + y^2$. Put in the following, at an address away from the main program:

```
1000 DEF FNpower(x, y) = x ^ 3 + y ^ 2
```

(Note that the space isn't required after DEF, but it *does* help legibility. Also you *can* have far more complex definitions than the above, even ones taking up several program lines like a normal PROCedure.)

Step 2

Whenever in your program you want to use your function, call the function with FNpower (x, y) or whatever. You don't need to use the same variable names each time—so these fragments both work:

```
10 INPUT "Values□", x, y
20 PRINT "Power□" FNpower(x, y)
110 INPUT "Numbers□", numb1, numb2
120 PRINT "Result□" FNpower(numb1, numb2)
```

(Note that, despite what some people say, you don't need to keep your function definitions apart from the main program—they don't interfere at all. It's just administratively more convenient to put them in their own little corner. Indeed you don't even need to RUN the DEF FNs that you wish to define, if you like to call them with PRINT FN . . . as direct commands; line numbers are *still* needed but not RUN.)

PROJECTS

1. Enter, study, use and understand this program:

Program 14: Factors

```
   10 MODE 5: COLOUR 130: COLOUR 1: CLS
   20 PRINT TAB(22) "**FACTORISATION**''' "(To find the fac
tors of an input number)"
   30 REPEAT
```

```
  40   PRINT''' "What is the number?": INPUT numb
  49   REM  ** Watch the white rabbit!
  50   IF numb<1 OR numb<>INT numb THEN PRINT "Enter positi
ve wholenumber only!": FOR wait = 1 TO 2500: NEXT wait: CLS:
 GOTO40
  60   CLS: PRINT' "The factors of"',numb ":"''
  70   FOR go = 1 TO numb/2
  80     IF numb = go*INT(numb/go) THEN PRINT go
  81     REM  ** OK mathematical genius....
  90     NEXT go: PRINT numb
 100   PRINT''' "Press RETURN for a  new run, or 0 to    st
op."''
 110   LET end$ = GET$
 111   REM  **  Wait for and test keypress
 120   CLS
 130   UNTIL end$ = "0"
 131 REM  **  Rogue value
 140 PRINT '''"Now learn all those...."'''
 141 REM  **  Joke
```

2. Now improve Program 14 as you see fit, making it as easy for the user as possible ('user friendly').
3. Or use the ideas in it to print out prime numbers up to the Electron's limit (if you have time . . .). A prime number is one whose only factors are 1 and itself.
4. An object falling from rest moves faster and faster to Earth. (I cheat and ignore air friction.) After t seconds its speed is 10t metres per second and it has fallen 5t metres. Write a program to input time of fall and display final speed and distance fallen, with suitable messages.
5. (A special for teachers and writers!) The 'required reading age' of a piece of text is given by a formula like this:

   $$AGE = 5 + 0.4(w/s + p/w)$$

   Here w is the number of words in the piece; s is the number of sentences; and p is the number of words of more than two syllables. Develop a text analysis program based on this.

10 Getting It Taped

By now, you will have typed a lot of programs into your micro, and lost them with NEW or when switching off at three in the morning. By now, indeed, you may well have got around to saving programs on to cassette and storing them for when you want to impress yourself or your visitors later on.

The main work in this chapter is, therefore, concerned with 'the cassette interface', so that the routines can be set out with any possible problems noted. Your early exploration of the Introductory Cassette will have helped you to overcome *these* barriers:

1. Finding a cassette recorder that works fairly reliably in this context.
2. Finding a suitable lead to connect this to the micro.
3. Getting the hang of LOAD ". . ." and CHAIN ". . .".

If you actually have to buy a cassette recorder, you can of course purchase the 'official' Acorn BBC one. At the time of writing this doesn't have a very good reputation, though I personally have never had any trouble with mine. You may, on the other hand, need to look for the cheapest mono cassette recorder you can find; try to get, if possible, these features:

remote control
tape counter
treble (tone) control

You *can* manage well without any of these (I do with most of the machines I use!), but they certainly help.

LOADING

This means getting a program that has been recorded (saved) on a cassette, back into the micro's memory. There are two commands here, as you know—LOAD and CHAIN (there are also the commands *LOAD and *RUN which you may need to use with some published software.)

Each should be followed by the name of the program to be loaded. The structure LOAD " " (or CHAIN " ") works, but may not be so reliable—use it if you like, to get the next recorded program, whatever it may be, into the computer memory. After all you'll *have* to use this simple form if you've forgotten the program's name . . .

You may use such structures as these (with CHAIN instead of LOAD) in each case:

1. LOAD " " [load the next program found]
2. LOAD "name" [load the program with that *exact* name, if found]
3. LOAD a$ [if a$ has been assigned]

LOAD can't be used in a program, so using (3) is a bit unlikely. CHAIN *can* be used in a program, so you can have things like this:

```
50 PRINT "Enter name of program wanted" ' ' "NAME", "CONTENT"
60 PRINT ' "1", "Breakout"
70 PRINT ' "2", "Household accounts"
80 PRINT ' "3", "Problems of quantum theory" ' '
90 INPUT prog$: CHAIN prog$
```

Yes, a program name can be a single number—as I noted on Page 35, it can be a set of one to ten keyboard characters.

I don't particularly recommend you to develop index programs like that unless your cassette contains only a small number of short programs—waiting time can be very long if you want the last one otherwise. Discs are better if you're rich enough to afford them, and then techniques such as this are ideal.

There are other differences between LOAD and CHAIN. The obvious one is that CHAIN sets the program running straightaway, a bit like LOAD". . .": RUN(R). Not *quite* the same though—LOAD clears the memory entirely, but CHAIN leaves some data that can be carried through from the old program to the new one. (*Some*, not a lot! But this is an advanced feature. It is used in the Introductory Cassette to carry the time through a series of programs.)

SAVING PROGRAMS

The command here is SAVE, which *must* be followed by the program name in quotes. (I mean, there are no tricks like SAVE " " and SAVE prog$ in this case—they wouldn't be much use anyway. SAVE cannot be used inside a program!)

The program name must not be '□' but otherwise can have any combination you like of letters, numbers, spaces and keyboard characters. The name can be 1–10 characters, but obviously long complex names could lead to later problems! (Recall that to LOAD a program by name, the name you use in LOAD "name" *must* be identical to that used when saving.)

If you find you can't re-LOAD programs SAVEd in the 'official' way, first listen to the recording by ear. If it is very quiet, very loud, muffled or distorted, or contains any hums or squeaks, you will need to check all aspects of your interface. Some possible problems may be solved during SAVEs by disconnecting the EAR lead (if you can); reducing the volume setting to just above zero; and moving micro and audio equipment away from the TV and the power leads. There's more on this subject coming next.

LOADING PROBLEMS

If you are able to load the Introductory Cassette programs, but your computer doesn't like your own recordings, try the following tips.

1. Play with the volume control.
2. Keep tone on maximum treble.
3. Move the micro and tape recorder as far from the TV as possible.
4. Make sure that the audio leads do not come close to any leads carrying mains current.
5. Load with the microphone link disconnected (if your machine allows that).

With some micros (but not the Electron as far as I know), overheating problems can make loading and saving erratic or impossible. A major difficulty can be that the saved signal is so bad that the computer can't understand it when trying to load. All sorts of strange messages appear on screen in this case; return the machine for modification.

One way to check for successful saving is to rewind the tape and enter the command *CAT (= catalogue). This prints updated details of all the programs met as the tape runs, and will show rubbish if any part of a program hasn't recorded properly. That's not a very satisfactory way to 'verify' cassette recordings of software, but it's pretty well the best that the Elk can do. (The main use of *CAT is of course to let you find the names of programs stored on a cassette (or disc), but you've written their names on the label, haven't you?)

If you *can't* load the Introductory Cassette, you sure have problems. There are various possible reasons for this, the two most important being that the lead you so expensively obtained is not in fact suitable, or that the cassette recorder has got what experts call a 'head alignment' problem. You could try borrowing a cassette recorder from a friend, especially one who has had no problem with it himself or herself, or if need be take the cassette recorder to a good audio maintenance shop and ask them to check head alignment and cleanliness.

ERROR MESSAGES

And now for something completely different—or have you been troubled with dire warnings from the Elk while going through the saving/loading stuff? We call those dire warnings—DATA, block?—*error messages* or reports.

It is fairly safe to say that a computer like the Electron can't make mistakes—it follows the instructions given (whether in ROM or typed in from the keyboard). But if it doesn't understand what you tell it, or you make a typing slip, it will gently tell you. Mind you, there are all sorts of errors that you don't find out about so easily—the computer understands the instructions, but they're not what you meant. (See Chapter 16.)

The *User Guide* gives (from Page 269 on) a fairly long list of 'error messages' that the machine is trained to supply in different situations. There may be others, but I haven't yet come across any. And it is possible that your machine may get so hot that it really does stop working properly after a while. This should not happen, of course—and I haven't heard of it happening with the Electron, but as I write only a few hundred of them have been sold. If you *do* have this problem, I'm afraid it may mean sending the machine back.

Particularly common error reports are as follows

Mistake You used an unrecognisable command or program statement. (In the latter case the report will tell you the line that needs checking.) For instance

"run" for "RUN"

or

"PR INT" or "PRONT" for "PRINT".

Syntax error You used a recognisable command or program keyword but what followed it doesn't follow the rules. For instance, "LIST n" is not acceptable to the Elk, even if 'n' has been assigned a value. (Many computers would accept this useful technique.) So the 'grammar' or *syntax* of the LIST command has not been obeyed.

In fact syntax errors are so common that many have their own messages, "no FOR" for instance. The *User Guide* does give at least brief notes, and on the whole messages are fairly clear, even if it is not always obvious what they're referring to! But I can't help you much more than this—I sometimes think that the Electron error reports are the subject of a whole book, rather than just of a page. Just one more tip, though. Electron error messages refer to the error line if it's in a program—but if your lines are long, it can be a devil of a job pin-pointing the problem. MORAL: Keep your lines short. That's a good moral anyway!

All the same, you should be aware of the useful structure "ON ERROR do such and such". As we've already noted, this blocks off the usual error routines for the rest of the program, or until it meets ON ERROR OFF (what a horrible phrase!). If there is an error of any kind thereafter it is handled as *you* decide. What you decide is rarely likely to

be as effective as what Acorn decided, and the main problem is that ON ERROR . . . blocks *all* error routines, so you should replace all of them to be safe. However, I used the technique in Program 13 to block program-stops due to trying to square root a negative number. But what I did would also block reports of any other possible errors made, for instance, as I then said, pressing ESCAPE. (Yes, Acorn deem pressing ESCAPE to be an error!) There'll be more on this subject in Chapter 16.

11 Coming to a Decision

Right at the beginning of this book (in Chapter 2) I went on about the fact that the Electron is able to carry out millions of actions in a second. The point then was (mainly) that this makes essential the 'stored program' concept. Now it is time to think about *decisions*—the single most important aspect of any programming system. There are two kinds of decision as far as we're concerned. Both make computers very powerful and yet easy to program. Both allow the micro to carry out many actions without too many instructions. I'll call these two kinds 'implied' decisions and 'open' decisions. (I made the words up, but I'm not alone in thinking the idea important.)

Without knowing it (I guess) you're very much used to *implied decisions*. They are needed in the *loop* structures given by the Electron's REPEAT . . . UNTIL and FOR . . . NEXT. Compare these two program fragments. Can you see that each expects (implies) that the micro can make decisions?

```
FOR go = 1 TO 1000          LET go = 0: REPEAT
(do such and such)          (do such and such)
NEXT go                     LET go = go + 1: UNTIL go = 1000
(do something else)         (do something else)
```

You can see both structures in Figure 11.1 below; it's very like Figure 7.1 on Page 39.

The decision part, in the diamond-shaped box, is essential. What the computer does is:

1. Assign the starting value to variable 'go'.
2. Do such and such.
3. Increase the value by 1 ('increment' it by 1).
4. Test whether 'go' has reached 1000
   (a) If not (FALSE) go back to step (2)
   (b) If so (TRUE) go on to do something else.

Each time round the loop, then, the computer carries out a test; the result of the test decides what happens next. The need for test + decision is *implied* in the loop structures REPEAT . . . UNTIL and FOR . . . NEXT (and the WHILE . . . ENDWHILE you may meet with other micros).

Here's a third program fragment that has just the same effect. This time the decision is not implied—it is out in the open. That's why I call it an *open decision*. We have met a number of examples of its main structure, IF . . . THEN, already.

```
100 LET a = 1
110 do such and such
120 LET a = a + 1
```

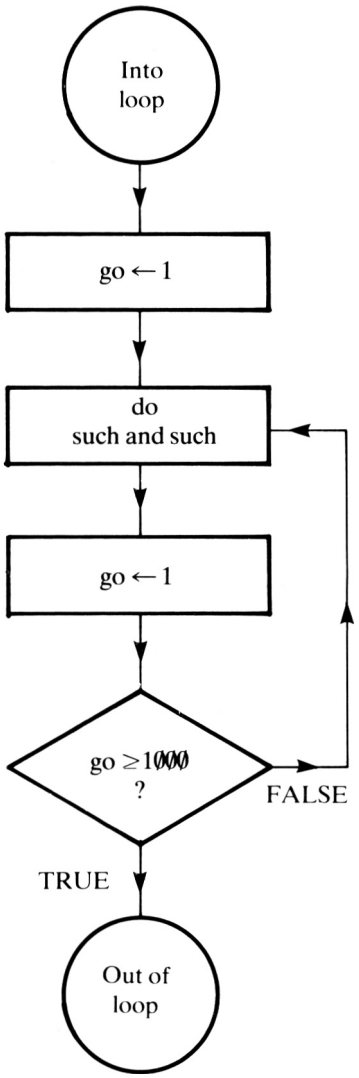


Figure 11.1

```
13Ø  IF a < 1ØØØ THEN GOTO 11Ø
14Ø  do something else
```

I hope you can see several things here. First, do you agree that it is in effect just the same as before? That, in other words, the diagram above describes it? What *this* approach does is simply bring the decision out into the open.

Something else different may be slightly less obvious—I've had to use line numbers. This is because of the GOTO statement—the micro has to know where to go. Programs without GOTOs and therefore without the need for line numbers are much 'nicer' than ones with. Unfortunately BASIC has a line number structure and there's nothing we can do about it. I do advise you to keep to a minimum the use of GOTO, however, as that will help to make your programs easier to follow.

IF . . . THEN

We can use this open decision structure in many more cases than just for looping. (In fact, why use it for looping? REPEAT . . . UNTIL and FOR . . . NEXT are much better!) It is, as I've said, very powerful and one of BASIC's most valuable structures. Here's how it works:

IF (something is true) THEN (do this) otherwise don't.

As soon as the computer gets to the IF it carries out the test 'is something (whatever it is) true?'. IF it is true THEN it carries out the next instruction. IF not THEN it doesn't.

Three points increase the power of this concept in the case of the Electron (and some other BASICs).

1. We can cater for the 'otherwise' situation using the keyword ELSE. IF (something is true) THEN (go here) ELSE (go there). Thus:

   IF score = 10 THEN GOTO 500 ELSE GOTO 600

2. The THEN doesn't have to have a GOTO after it: any keyword can do:

   IF score = 10 THEN PRINT "Excellent!" ELSE PRINT "Keep trying!"

3. The multi-statement approach works, using colons:

```
IF score = 10 THEN PRINT "Excellent!": LET bonus = bonus + 1:
           PROCreward ELSE PRINT "Keep trying!": IF bonus > 0
           THEN LET bonus = bonus -1
```

We don't need all those spaces in that last example: as usual I put them in to make what's going on clearer. *It is very easy to become confused in IF . . . lines.* Figure 11.2 shows what the final example does. Complex? Complex! Please take *great* care with IF . . . lines. Otherwise you'll spend ages trying to find out why your precious program doesn't work properly. You have been warned!

To make matters worse, you don't actually need the THEN in Electron BASIC, but there *must* be a space:

IF A > B A = B ELSE B = A

Confusing, yes? (I've left out LET to confuse you more.) To make matters *still* worse you don't need a GOTO before a line number in IF . . . statements:

IF A > B THEN 600 [this is an 'implied' GOTO]

But you can't write

IF A > B 600

or

IF A > B PRINT "Bigger": 600

I'm really telling you all these last points so that you can recognise them in program listings. I'd strongly advise you always to use THEN and GOTO in your own programs—it makes them easier to read, less open to error and less hard to check. And here's a final, *very* important warning.

This works:

```
100 IF ans$ = "CAT" THEN PRINT "Good"
110 PRINT ' "Now try the next one."
```

Whatever the value of ans$, the message "Now try . . ." appears. *This* multi-statement line looks as if it should do the same:

```
100 IF ans$ = "CAT" THEN PRINT "Good": PRINT ' "Now try the next
one."
```

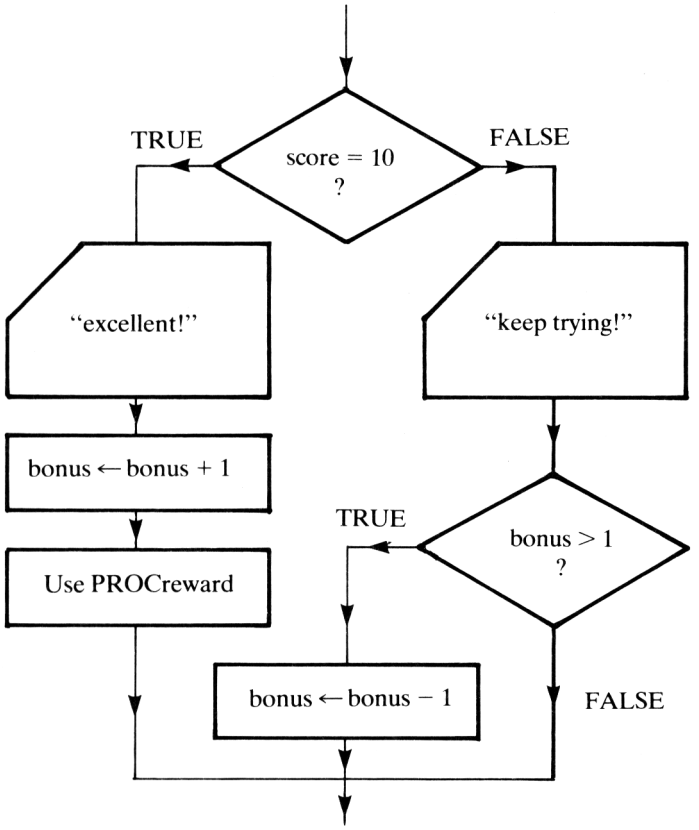


Figure 11.2

But the message comes on screen *only* if ans$ = "CAT".
The colon multi-statement idea needs special care in IF lines. This is because in such a line, this is what the computer does.

1. Carry out the test after IF.
   (a) If the test gives a TRUE answer, then carry out all statements until an ELSE appears . . .
   (b) If the test gives a FALSE answer, scan the line for an ELSE, then carry out all statements after it . . .
2. Go to next line.

Check that with my second "CAT" example. So—to repeat, yet again, *it is very easy to become confused in IF . . . lines!*

The following program should help you to understand the versatility—and dangers—of IF. Enter it, try to predict the outcome of different values of A, and see if you're right. And if you're keener still, re-write it so that it is much clearer. Also can you get it into two lines?

Program 15: IF what?

```
10 LET a = INT(RND(5))
20 IF a = 1 THEN PRINT 1
30 IF a = 1 THEN GOTO 10
40 IF a = 2 THEN PRINT 25
50 IF a = 2 THEN CLS
```

```
 60 IF a < 3 THEN PRINT a*100
 70 IF a = 3 THEN PRINT "a=4"
 80 IF a > 2 THEN IF a < 5 THEN FOR b = 1 TO 1000: NEXT b
 90 IF a*a < 20 THEN PRINT a " squared is ";a*a
100 IF a = 4 THEN PRINT PI
110 IF a = 4 THEN GOTO 10
120 IF a < 3 THEN STOP
130 IF a = 5 THEN PRINT a "mess"
140 GOTO 10
```

MORE IFS AND BUTS

Quite often we need an IF . . . THEN IF . . . structure. 'IF it is raining THEN IF I have an umbrella THEN I'll use it ELSE I'll take shelter.' The same is true of computing as of real life—'IF the score is ten THEN IF there've been ten goes THEN congratulate and finish ELSE give more practice.' Sometimes we may need even more complexity—IF . . . THEN IF . . . THEN IF . . . THEN . . . ELSE IF . . . THEN IF . . . and so on. I shan't try to give an example; I'm sure you take the point.

We can re-word the raining sentence like this; 'IF it is raining AND IF I have an umbrella THEN I'll use it ELSE I'll take shelter.' And we can do the same in BASIC. But I hope you can already see dangers—which IF did the ELSE refer to? I'll come back to that, but be warned in advance.

> "IF the score is ten AND IF there've been ten goes THEN congratulate . . . and so on."

The AND here is a useful keyword (strictly it's called a *logical operator*).
Here's what the line would actually look like:

> IF score = 10 AND goes = 10 THEN PRINT "Full marks" ELSE PRINT "Here's another one!": GOTO somewhere

Take a look at Program A3 (at the end of the book); this shows how this AND business works in a sort of game. I *could* have put all the IFs into one complex IF . . . THEN . . . ELSE etc. etc. line, but it *would* have been complex, wouldn't it?

Another logical operator of value here is OR. With care (again!) you can compare it to the English 'or'—IF it is raining OR it is snowing AND I have an umbrella THEN I'll use it ELSE I'll take shelter. So—in BASIC:

> IF score = 10 OR score = 9 AND goes = 10 THEN PRINT "A good score" ELSE . . . and so on.

But you *can* see the rocks of complexity again, can't you? Fortunately we can use brackets to ensure the right priority for the computer if we wish:

> IF (score = 10 OR score = 9) AND goes = 10 THEN . . .

or even (to be completely sure)

> IF ((score = 10 OR score = 9) AND goes = 10) THEN . . .

Structures like this that use only one IF do not lead to the ELSE problem I mentioned above—if there's only one IF then the ELSE must apply to it. If there's more than one IF, the ELSE applies to the *last* one. Indeed you *can* nest them: IF this THEN IF that THEN such and such ELSE so and so ELSE the other. But *how* complex!

CONCLUSION

IF is *very* important, and in Electron BASIC is very flexible. However, the flexibility leads to all kinds of possible dangers—TAKE GREAT CARE WITH IF!!

If you're human, THEN you're by now probably getting a bit lazy with the projects in these pages. But please take some time over the ones that now follow, ELSE you're likely to have many problems in future . . .

PROJECTS

1. Draw the detailed flowchart section that would show these program lines:

```
  100 LET brown = 0: LET red = 0
  110 FOR go = 1 TO 50
  120    INPUT''' "Colour",answer$
  130    IF answer$ = "BROWN" THEN LET brown = brown+1 ELSE I
F answer$ = "RED" THEN LET red = red+1 ELSE IF answer$ = "WH
ITE" THEN LET brown = brown+1: LET red = red+1 ELSE PRINT "O
nly BROWN, RED or WHITE please!": GOTO 120
  140    PRINT "Brown", brown' "Red", red' "White", brown+red
  150    NEXT go
```

2. Code the loop shown below in three different BASIC ways.

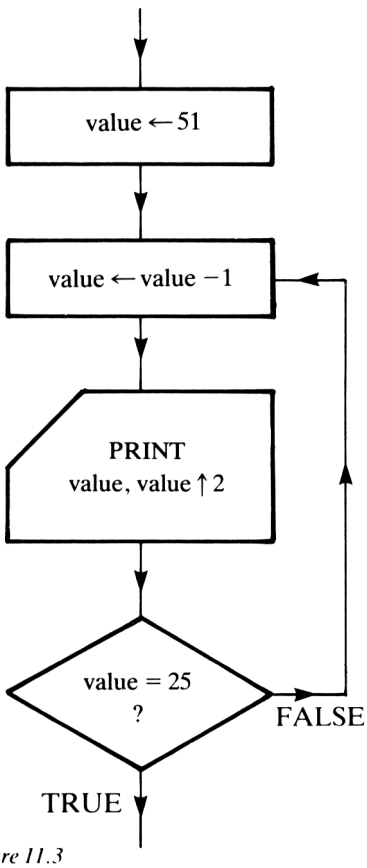


Figure 11.3

3. Write your own program like (but better than!) Program 15.

12 Picturing Programs

In the last chapter, in which I had a bash at decision-making, the need became urgent for showing what's going on in pictorial form. The kind of diagram I then used to show the action of various structures has also appeared earlier in the book—it is called a *flowchart*.

It is important for a programmer to be able to devise and to read flowcharts as it is for an electrician to deal in circuit diagrams. In fact the concepts are very much the same in both cases:

1. The diagram lets one see very quickly what's going on;
2. The diagram is easier to check than the actual creation (circuit or program);
3. The diagram is fairly universally understandable.

Compare the circuit in Figure 12.1 with the diagram beside it. Which better shows the viewer the structure?

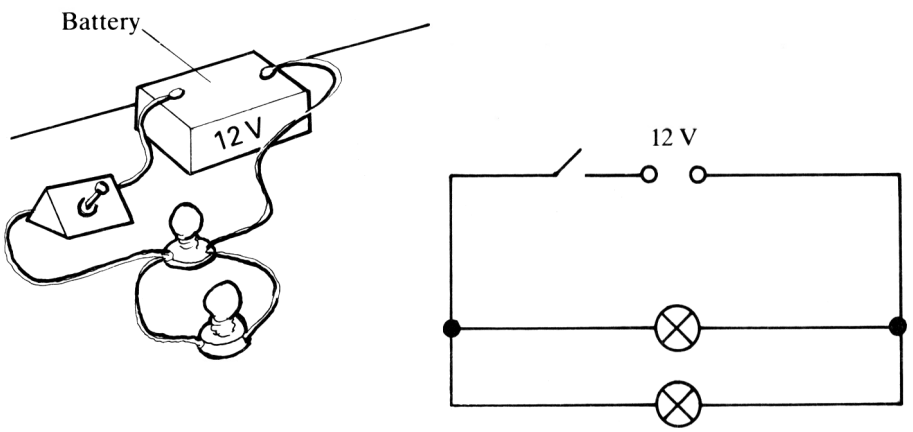


Figure 12.1

In just the same way the symbolic London Underground map is much easier to use than a real one would be. And in just the same way too a flowchart shows structure much better than a BASIC listing. It shows where you start and where you get to, and the different routes (branches) on the way.

Flowcharting is very simple. All you need to know is the set of symbols used and rules to be followed. (Mind you, I must admit that the language of flowcharting suffers a bit from dialect problems!) Anyway, here is the concept as much as we need to know it.

THE LANGUAGE OF FLOWCHARTING

I shall set out symbols and rules together. Here goes.

Rule 1 A flowchart must have one beginning and (preferably) only one end.
SYMBOLS—the START and STOP 'boxes' are shown in Figure 12.2.

Figure 12.2

Rule 2 The direction of flow in a chart is from START at the top to STOP at the bottom. The normal direction of horizontal flow is to the right. Use arrows for all other directions.
SYMBOLS—boxes (action stages) are joined by straight lines, with arrows *at least* where normal directions are not followed. See Figure 12.3.

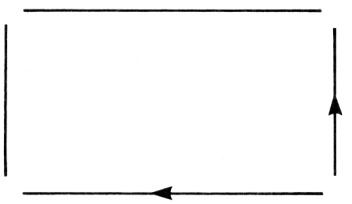


Figure 12.3

Notes: I think it wise to use more arrows than the minimum. Also, if a chart covers more than one sheet of paper, use connector boxes like those in Figure 12.4.

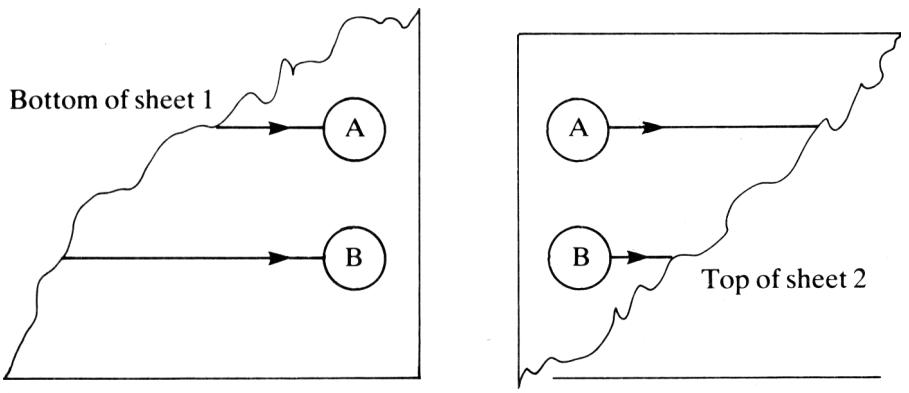


Figure 12.4

Rule 3 Show each stage in the stucture with a 'box' in the flowchart.
SYMBOLS—we need boxes to show INPUT, OUTPUT, DECISION and any other kind of *action*. They are shown in Figure 12.5.

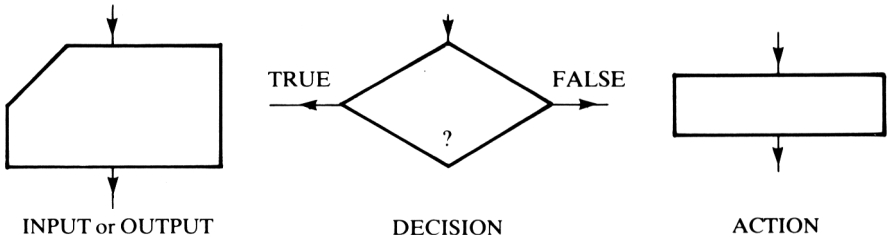


Figure 12.5

Notes: These symbols are where some people differ! Also there are plenty more special symbols that posh folk in industry and commerce may use, but this is quite enough to get by on.

And that's it. Easy, n'est-ce pas?

When I teach science (which is officially my job . . .) and get on to circuit diagrams, I give practice in two ways. Firstly, given a circuit description, prepare the diagram; secondly given a diagram, describe or prepare the circuit. Our pupils and students need to be very familiar with circuit diagrams. And it's just as true of flowcharts. They are very useful indeed—so there are some projects on them at the end of the chapter. But in what ways *are* they so very useful?

USING FLOWCHARTS

It used to be traditional (I mean, a few years ago!) to expect programmers to develop flowcharts before starting to write the program itself. There would then be two stages. The *outline flowchart* would show, in only a page or two, the broad stages of the project. Then a *detailed flowchart* would set out what the program would do almost line by line.

Nowadays, like many other teachers, I feel that there is no real need for detailed flowcharts. The program itself should be so well laid out that it would be quite clear.

Some people go further and suggest that the outline flowchart should be drawn *after* the program is finished rather than before it starts. But I don't agree with that. I advise you *always* to put your ideas down as an outline flowchart before starting work on anything more than a very simple program. It doesn't need to be beautifully neat, but it *does* need to be clear. The flowchart habit will save you a lot of time in the long run . . .

DOCUMENTATION

And now we come to another important piece of advice. It is that you *keep adequate records of your programs*! By this time you are probably beginning to build up what may become a fairly extensive library of Electron program cassettes. Get organized now!

1. Do not keep more than four lengthy programs on one cassette (two on each side). Make and keep second copies of anything you'd hate to lose—cassette recordings can be ruined, by you, the cassette machine, or the dog.
2. Number the cassettes (whether they are ones you've recorded or ones you've bought). Code duplicates as (a) and (b). Thus

   5a. ELK INVADERS—master copy. 5b. Copy for daily use.

3. Buy a loose-leaf file and keep in it an index plus, in number order, all the papers associated with each cassette.
4. Store the cassettes neatly, again in number order, and keep the file near them. The cassettes should not be exposed to direct sun, high or low temperatures, dust or magnetic fields. Look again at the picture on Page 6!

If you start such a system now, when you have only a few cassettes, you'll be glad of it when you have dozens. (I learned the hard way, but what I've described is what I try to do now with the many hundred program cassettes that try to submerge my work areas.)

Anyway, what about this paper-work in the file, this *documentation*? You've got to provide for two situations, so there are two kinds of documentation—user documentation and programmer documentation.

User documentation

User documentation includes everything associated with the program that helps anyone using it to run it without problems. It is, in other words, the set of instructions.

I think that the only paper instructions needed by the user (of anything but a complex program) should be how to load it—though a description of one or two lines in the index is obviously required. Everything else should be in the program itself. Use lines like this:

```
50  PRINT ' "This program does such and such."
```

or

```
50  PRINT ' "This is a game where you have to do so and so."
60  INPUT ' ' ' "Do you want INSTRUCTIONS (Y or N)", a$
70  IF a$ = "Y" OR a$ = "y" THEN PROCinstr   (We'll deal with PROC later)
80  [get on with the program]
    . . . . .
4990  STOP
5000  DEF PROCinstr
5010  CLS: PRINT ' ' ' "Here is what you do . . ."
    . . . . .
5090  CLS: ENDPROC
```

There are two major advantages of putting user instructions into the program. Number 1—no bits of paper to lose. And Number 2—it'll force you to keep the instructions simple, short and clear! User-friendliness—remember?

Programmer documentation

Now for something completely different. What this kind of documentation has to do is help you (or someone else) to improve the program in the future. What do you mean?—'my programs can't be improved.' Wrong!—just you wait and see. For a start, each user will (unless shy) give you extra ideas. And then, in a few months, your growing experience will show you new and better ways to do things. *Programmer documentation makes later improvement easy*. To some extent you can put this kind of information in the program. BASIC has the useful REM (= remark) statement just for this purpose. Any line starting with REM appears when you LIST, but is ignored when you RUN. Let me add REMs to the last program fragment to show what I mean:

```
10  REMARKABLE GAME BY AMINA BLOGGS SEP 1983!!!
49  REM introduction
```

```
  50  PRINT ' "This is a game where you have to do so and so"
  60  INPUT ' ' ' "Do you want INSTRUCTIONS (Y or N)", a$
  69  REM a$ is Y/N answer: Y sends control to PROC at line 5000.
  70  IF a$ = "Y" or a$ = "y" THEN PROCinstr
  78  REM * * * Program start * * *
  79  REM = = = ==== ======= = =
  80  . . . . . .
4989  REMember to delete next line
4990  STOP
4999  REM instructions
5000  DEF PROCinstr
5010  CLS: PRINT ' ' ' "Here is what you do . . ."
      . . . . .
5089  REM—Return to main program, line 70
5090  CLS: ENDPROC
```

Fine. Well, that's all clear now. So why not just use REMs? Why bother with paper that's likely to get lost?

In short, you *needn't* bother—with *simple* programs. But if your program is more than a few dozen lines you *do* need paper. After all, an adequate number of REMs will take up quite a lot of memory and also increase SAVE/LOAD times. And it's not easy to use them when they're scattered through a number of screens of listing. So, think about keeping the following in your paper file:

1. Program name, length, date, author, cassette number(s).
2. Program description, and source of idea.
3. List of all variables used, with their meanings and any comments.
4. Outline flowchart with, added, the line numbers covered by each box.
5. Notes of any special tricks you used.
6. Date and details of later amendments.
7. Ideas for future development.

Do all that nicely (even if not very neatly) and one day you may be able to add:

8. Date and place of publication: price; royalty!

PROJECTS

1. Choose two or three of the programs earlier in this book and one of the simpler ones at the end—and prepare a careful outline flowchart. Programs 13 and A3 are *real* tests! (Keep your answers, by the way.)
2. Study the flowchart in Figure 12.6 and then describe the idea in words.
3. . . . and then, if you're keen, write a program to express it.
4. Add REMs to one or two of your own programs and try to work out how many there *should* be.
5. Prepare a full two sheets of program documentation for one of your less simple programs.
6. Try to write a program based on one of your answers to question (1). Compare the result to the original and decide how good your flowcharting is!
7. Organize your software collection as I've suggested, including the file, before it's too late . . .

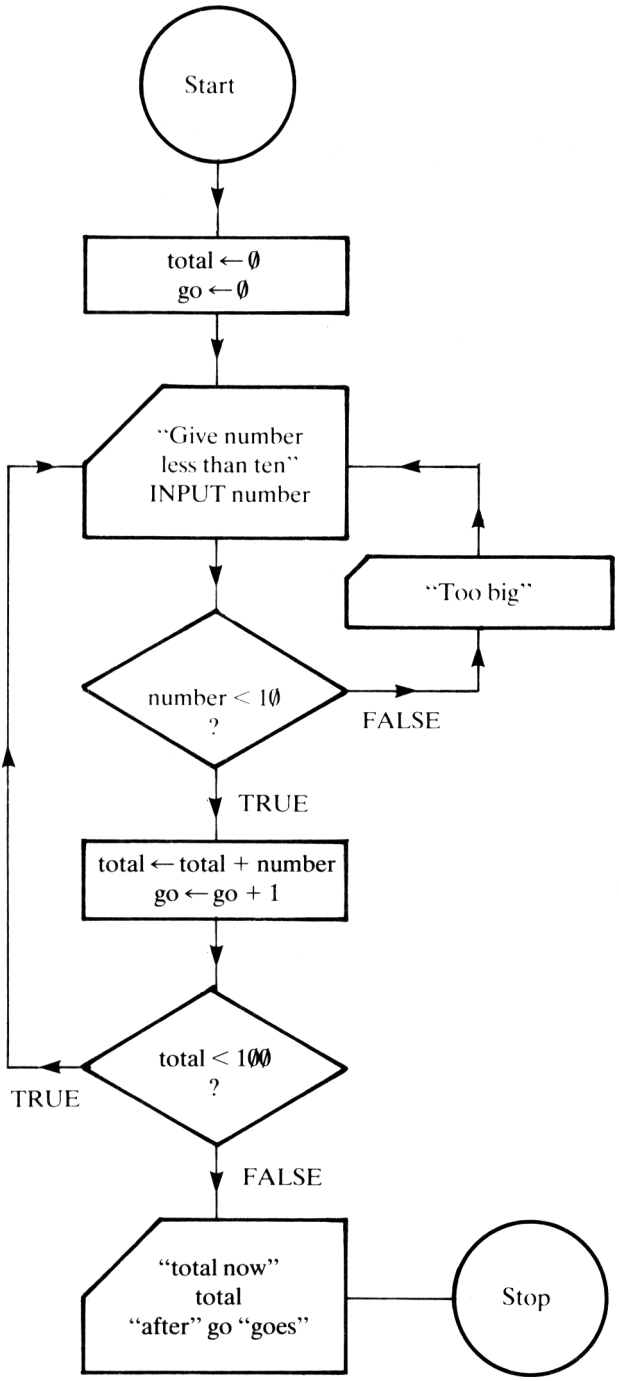

Start
total ← Ø
go ← Ø
"Give number
less than ten"
INPUT number
"Too big"
number < 1Ø
?
FALSE
TRUE
total ← total + number
go ← go + 1
total < 1ØØ
?
TRUE
FALSE
"total now"
total
"after" go "goes"
Stop

Figure 12.6

13 Sharing The Proceeds

Now that our Electron programs can include open decision-making, they can gain a more complex structure. (And that's why I took time off just now to look at flowcharting.) Electron BASIC offers two facilities for keeping things simple in the more advanced programs we may now work on. The word *subroutine* covers both. The facilities are of great value, and I am delighted to be able to use them in the rest of the programs in this book.

A subroutine is any section of a program (routine) which has a clear single purpose. I want to explore the background theory in more detail in the next chapter, so let me just give a summary now. The most efficient way of developing a program is to break the initial concept down into a number of chunks. (Recall the outline flowchart?) Each chunk (or module) has one function and can be coded into a subroutine to go in the program as a whole. If that's not clear, take a look at Figure 13.1. Each box shows a module.

So that's subroutines then—chunks of a complete program each with their own job to do. What most folk mean by subroutines are chunks that appear separately in the listing from the main program stream. Figure 13.2 shows this. What the main program then does is to 'call' on a given subroutine whenever it's needed. We describe such subroutines as *closed*.

This approach has two advantages. The first is obvious—in a program of any length you are likely to find that the same module has to be used several times (SUB A in Figure 13.2). Then it need appear only once in the listing, yet you can call it as often as it's wanted. The second advantage is less obvious. It is that the contents of a subroutine are particularly easy to check and correct, without involving the main program. This can save a huge amount of time and trouble.

GOSUB . . . RETURN

I said Electron BASIC has two ways of giving closed subroutines. First I'd like to discuss the one that's standard to all BASICs. Because the Elk has a non-standard one too, the way it deals with this particular one is not very advanced.

There are two new keywords here. To call up a subroutine, use GOSUB n, where 'n' is the number of the first line of the sub-routine called. To tell the micro that the subroutine is finished, use RETURN. Control then goes back to the line after the call in GOSUB. I try to show this more clearly in Figure 13.3.

When the computer meets GOSUB n it stores in memory its current position in the program, and then goes off to line 'n'. When the computer meets RETURN it fishes around in memory to find where to return to, and back it comes. Just like using a bookmark to keep your place while you're looking at the pictures in the middle.

The section of memory that stores the current position can get pretty busy. This is because you can 'nest' subroutines as much as you like—a subroutine can call a second which can call a third and so on. Just as with nested IF statements and loops, though, things can get tricky in practice—you need to take care.

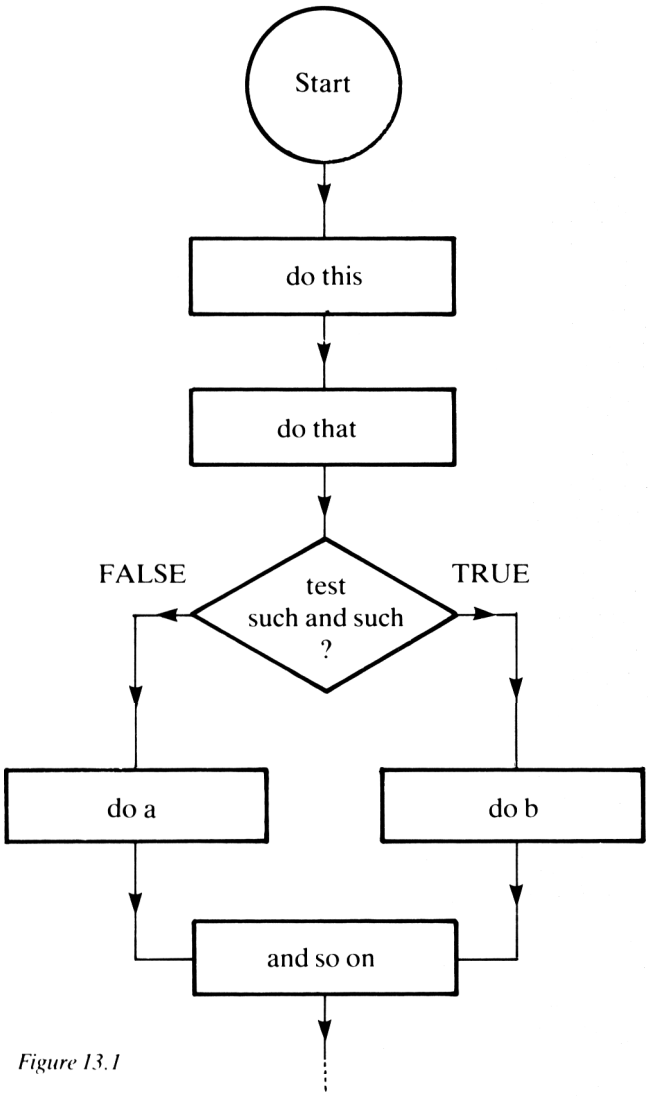


Figure 13.1

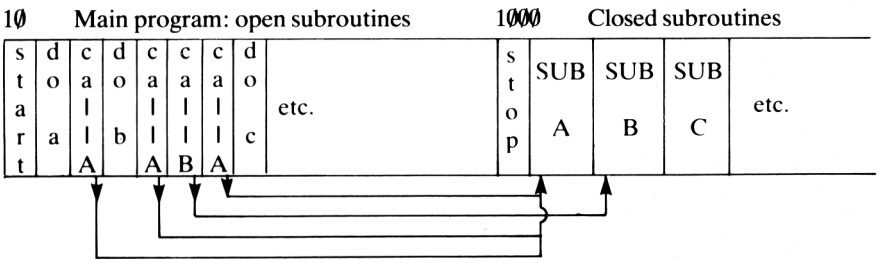


Figure 13.2

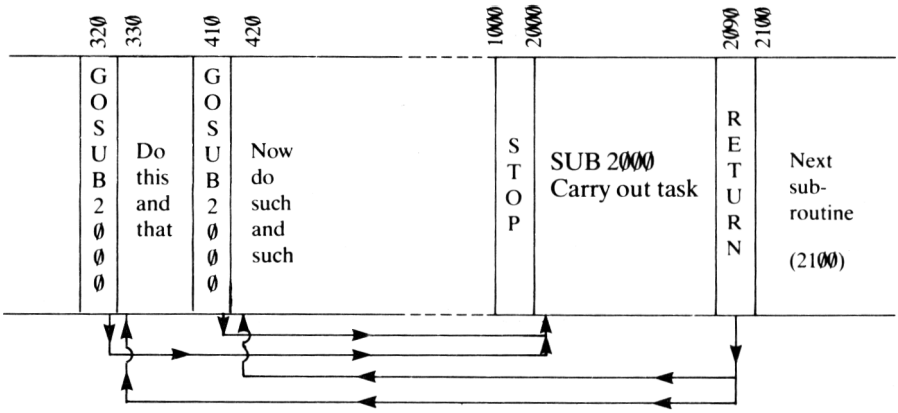


Figure 13.3

It's a useful structure, the closed subroutine. Get used to it, especially when your programs need the same lines time and time again. Program 16 is an example. Study it! I've put a sketch of the flows of control after the listing (Figure 13.4). There is no FUNC way to get GOSUB; indeed it is interesting that there *is* FUNC + G to give GOTO, but nothing for GOSUB—perhaps that says something about Acorn's real attitude to the importance of GOTOs! You can use the abbreviation GOS. for GOSUB.

Program 16: GOSUB screen fill

```
 10 MODE 5: COLOUR 129: COLOUR 3
 20 GOSUB 400: REM  ** wait
 30 PRINT' " Enter one letter & RETURN."
 40 INPUT''' a$
 41 REM  **  Useful trick there!
 50 GOSUB 300: REM  ** fill
 60 GOSUB 400: REM  ** wait
 70 PRINT' "What's your name?"'''
 80 INPUT name$
 90 LET a$ = name$+" "
 91 REM  **  Adding strings
100 GOSUB 300: REM  ** fill
110 GOSUB 400: REM  ** wait
120 PRINT' " Now give any set ofcharacters you like!"
130 INPUT''' char$
140 LET a$ = char$
141 REM  **  Save a line with INPUT a$, BUT see later
150 GOSUB 300: REM  ** fill
160 GOSUB 400: REM  ** wait
170 LET a$ = "Thanks, "+name$+"! "
171 REM  **  Adding more strings!
180 REPEAT
190   GOSUB 300
200   GOSUB 400
210   UNTIL FALSE
220 STOP
221 REM  **  Not needed, but good habit
299 REM  **  Here's SUB fill
300 CLS
310 FOR rpt = 1 TO 640/LEN a$
311   REM  **  LEN a$ gives no of characters in it
320   PRINT a$;
321   REM  **  NB semi-colon
330   NEXT rpt
331 REM  **  Put REMs outside loop for best speed
340 GOSUB 400
341 REM  **  Nested subroutines
350 RETURN
```

```
351 REM  **  Back to main routine
399 REM  **  Here's SUB wait
400 FOR w = 1 TO 2000: NEXT w
410 CLS
420 RETURN
```

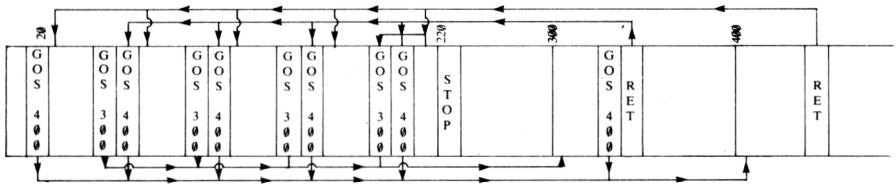


Figure 13.4

Here are some points about the GOSUB . . . RETURN business that you should note.

1. You cannot use structures like this in Electron BASIC: INPUT "Give a number between 1 and 4", number: GOSUB number *100 followed by relevant subroutines (each with RETURN) at lines 100, 200, 300 and 400.
2. The keyword ON *can* give this effect, however: INPUT "Give a number between 1 and 4", number: ON number GOSUB 100, 200, 300, 400. Here if 'number' is 1, we go to line 100; if it is 2, we go to line 200, and so on. Careful trapping is needed, to avoid problems if, for instance, 'number' ≠ 1, 2, 3 or 4.
3. It is much nicer to 'name' your subroutines. In the case of the above program, for instance, you could have 5 LET fill = 300; LET wait = 400, and wherever 300 appears in the listing, you can replace it with 'fill', and in the same way you can replace 400 with 'wait'. The only disadvantage with this system is that you will have to change the assignments in line 5, should you change the line numbers at which the named subroutines begin.
4. You can enter a subroutine in the middle if you have the need. Follow this fragment through:

```
  120 INPUT "Give a number between 1 and 4."'''' numb: IF nu
mb<1 OR numb>4 THEN GOTO 120 ELSE IF numb<>INT numb THEN GOS
UB 1000 ELSE GOSUB 1030
  121 REM  **  Here is a 'MUG-TRAP'!
  990 STOP
  999 REM  **  INTing part of sub
 1000 PRINT'' "I can work only with whole numbers."''' "You
gave the value "; numb " --"
 1010 LET numb = INT(numb+0.5)
 1020 PRINT "so I shall take it as "; numb "."
 1029 REM  **  main part of sub
 1030  etc
 1100 RETURN
```

Did you follow? The subroutine *really* starts at 1000, but the first lines deal specially with cases where the user didn't enter a whole number. If he or she *does* enter a whole number we don't want to use those lines, so we GOSUB straight to 1030. This is a useful trick. So too is to have several exits from (several RETURNs in) a subroutine. But *don't* jump out of a subroutine with GOTO—it will clog up the memory something chronic.

5. Keep your closed subroutines well away from the main program. The best plan is to put them after line 5000 (say) with STOP at line 4990 to prevent accidental entry. There's one rather advanced point here, though—the higher a subroutine's starting line number, the longer it takes for the routine to be found. This is because when the

micro's told to GOSUB n, it starts at the beginning of the program in its search for the line with that number. So if you want the highest speed, you can do this instead:

10 GOTO 1000

10–990 subroutines

1000 start main program.

Note that you cannot tell the Electron to RUN from any line other than the one with the lowest line, unlike some other micros.

6. Some people strongly recommend that you start each subroutine with a REM giving its title/function, or its name if it has one. I don't do that myself because I always remove REMs from the 'final' version of programs and if this is done with the Electron the programs would no longer work.

OK? Let me summarize what we've had so far.

1. A subroutine is a section of a program with a single function.
2. A closed subroutine is one kept apart from the main program, called when needed by GOSUB n and closed by RETURN.
3. The main use of a closed subroutine is to carry out repeated tasks without having to repeat the program lines.
4. A major subsidiary benefit is that you can use GOSUB n as a command in order to test the subroutine's workings as a single unit.

Straightforward! But now let's look at the Electron's second closed subroutine structure, the 'procedure'. It's *much* nicer.

PROCEDURES

We can use procedures exactly like subroutines. Here are the basic differences.

1. Refer to a procedure by name rather than by line number. (Though the name *can* be a number if you'd ever want to use that.)
2. Call the procedure with PROCname (no space) instead of GOSUB n.
3. Start the procedure with DEF PROCname (that space is optional).
4. Close the procedure with ENDPROC instead of RETURN.

That can all best be shown by an example—so here is Program 16 again, but using procedures rather than subroutines. Exactly the same diagram would apply to it (Figure 13.4), and I'll try to use the same line numbers so you can quickly edit it from the weedy GOSUB version.

One little tip before you get onto that, though. By now we're often dealing with programs with lots of lines, so that they more than fill one screen. I've told you how to use CTRL + N when you list in order to get 'paged mode', and here's how to have the same effect within a programmed BREAK key.

*KEY 10 "OLD¦M VDU14: LIST¦M" (R)

Posher, and simpler is

*KEY 10O.¦M¦NL.¦M (R)

Program 17: PROCscreen fill

```
10 MODE 5: COLOUR 129: COLOUR 3
20 PROCwait(1000)
30 PRINT' " Enter one letter & RETURN."
40 INPUT''' a$
50 PROCfill(a$)
60 PROCwait(2000)
70 PRINT' "What's your name?"'''
80 INPUT name$
100 PROCfill(name$+" ")
```

```
110 PROCwait(3000)
120 PRINT' " Now give any set ofcharacters you like!"
130 INPUT''' char$
150 PROCfill(char$)
160 PROCwait(3000)
180 REPEAT
190   PROCfill("Thanks, "+name$+"! ")
200   PROCwait(2000)
210   UNTIL FALSE
220 STOP
221 REM  **  Still not needed, but good habit
289 REM  **  Here's SUB fill
290 DEF PROCfill(fill$)
300 CLS
310 FOR rpt = 1 TO 640/LEN fill$
320   PRINT fill$;
330   NEXT rpt
340 PROCwait(1000)
341 REM  **  Nested procedures
350 ENDPROC
351 REM  **  Back to main routine
389 REM  **  Here's SUB wait
390 DEF PROCwait(limit)
400 FOR w = 1 TO limit: NEXT w
410 CLS
420 ENDPROC
```

If you compare with care the bits in Programs 16 and 17 that concern closed subroutines, you'll see that the procedures approach is more complex. So why use it?

There are two reasons. The first I've pointed out already. In Electron BASIC, GOSUB is best followed by a line number while PROC is easy to associate with a name. So using procedures lets us name the closed subroutines very nicely. That's useful both when writing programs and when reading them. On the other hand numbers let you find a given sub-routine in a long listing quickly. Needles in haystacks can be easier than DEF PROC in a program of hundreds of lines.

The second reason is more advanced; it's to do with how variables are treated in the main program and procedures. I'm not going to say much about it because this *is* advanced, but we need to take a little look. Check with Programs 16 and 17 to see what I mean in practice. And note particularly the REMark of line 141 of Program 16. In the GOSUB approach you must:

1. Define all the variables you're going to need before going to the subroutine.
2. Make sure that any other variables used in the subroutine have different names from those in the main program.

Both these restrict the use of GOSUB a great deal—but both restrictions are far less severe in the procedures approach. Look at this set of program fragments, using GOSUB and PROC respectively. I have compressed the listings to fit them nicely on the page, but that is a good opportunity for you to compare the readability of compressed listings with that of ones with plenty of extra spaces!

GOSUB fragments

```
 50  a=10:b=20:c=14
 60  GOSUB500
100  INPUT"x,y";x,y
110  a=x:b=y
120  GOSUB500
490  STOP
500  REM sub
510
520  FORd=1TO10
530      PRINTd,d*(a^2+b^3)
540      NEXTd
550  RETURN
```

PROC fragments

```
c=14
PROCans(10,20)
INPUT"x,y";x,y

PROCans(x,y)
STOP
DEFPROCans(a,b)
LOCALc
FORc=1TO10
    PRINTc,c*(a^2+b^3)
    NEXTc
ENDPROC
```

As I said, the above material is compressed. It also won't run, so please don't try! The reason it won't run as it is is that I have not put line numbers on the right hand side. If you look carefully, there is absolutely no reason to have line numbers there, in fact, as the use of procedures is more 'structured', and structure means that line numbers aren't required. Unfortunately BASIC *does* need line numbers, so you would have to put them in even if they weren't required.

I know this is a bit artificial but the main program each time needs the variable 'c' and needs to call on the subroutine with various values of 'a' and 'b'.

As I said before, in the GOSUB approach, you must:

1. Define all the variables before using the subroutine (lines 5Ø and 11Ø).
2. Make sure the other subroutine variables differ from those in the main program (here we need the extra variable 'd').

Procedures may be a bit more complex, but . . .

1. Variable values can be passed directly by putting them in brackets after the procedure name—lines 6Ø and 12Ø for calling, and line 5ØØ for defining. We then call the variables 'parameters'.
2. These, and other subroutine variables, if defined as LOCAL (line 51Ø), will not interfere with those in the main program.

OK, procedures *are* more advanced than using GOSUBs—but they *are* more flexible and I suggest you get used to them from now on. Here then is a statement of their use extended from the one on Page 78.

1. Refer to a procedure by name rather than by line number. The name may include, in brackets, the variables (parameters) to be passed between main program and procedure. Thus the full name may be "result (a, b$)", where 'result' is the title and there is a numeric parameter and a string one.
2. Call the procedure with PROCname, with, in brackets, the current values of the variables to be passed. Thus PROCresult (5, "The result is ") or PROCresult (value, message$).
3. Start the procedure with DEF PROCname, with, in backets, the subroutine's names for the variables passed:

   DEF PROCresult(p, d$)

   Then use LOCAL to tell the computer about other variables only to be used in the subroutine:

   LOCAL a, square, a1$

   You can use these variable names in the main program as well without fear of any upset.
4. Close the procedure with ENDPROC.

SUBs are simple: PROCs are powerful. Use the latter even when your needs are simple!

An important note

I think some people take the theory of program structure too far, and construct their main programs entirely out of procedure calls. Program 17 would then start something like this:

```
PROCset up
PROCwait
PROCinput(" enter one letter & RETURN.")
PROCfill
PROCwait
PROCinput(" What's your name?")
```

```
PROCfill
PROCwait
PROCinput(" Now give any set of characters you like!")
PROCfill
PROCwait
REPEAT
PROCfill
PROCwait
UNTIL FALSE
REM end main program
STOP
REM procedures
DEF PROCsetup
. . . and so on
```

Of course in practice your Electron would need line numbers even in such a highly structured program as that (though if you dared, you could put the whole of the main program as line 1). However I do feel that the above is rather like making a haystack out of needles—no room for the hay. (I've even seen main programs like this PROCstart: PROCprog:PROCfinish . . . with lots of complex nested subroutines to struggle through. Just think if a story was like that (PROCchapter1: PROCchapter2 . . . DEF PROCchapter1: PROCpara1: PROCpara2 . . .).

I think I've gone almost too far in this direction in the next program! Here is a pretty little thing that demonstrates the use of procedures.

Program 18: Billiards

```
  10 MODE 5: PROCsetup
  20 REPEAT
  30   PROCball1
  40   PROCball2
  50   PROCball3
  60   UNTIL 0: REM ** Short for U.FA.
  69 REM  ** Lines starting * are another kind of REMs; you
can leave them out too!
  70 *
  80 *
  90 *  REM Procedures now
 100 *
 110 *
 120 DEF PROCsetup
 130 COLOUR 129: CLS
 140 LET xx1 = 1: LET yy1 = 1
 150 LET xx2 = 2: LET yy2 = 2
 160 LET xx3 = 3: LET yy3 = 3
 170 LET x1 = 8: LET y1 = 13
 180 LET x2 = 10: LET y2 = 15
 190 LET x3 = 12: LET y3 = 17
 200 VDU 23 224 0,28,62,127,127,127,62,28
 210 ENDPROC
 220 *
 230 *
 240 *
 250 DEF PROCplot(x,y,p$)
 260 PRINT TAB(x,y) p$
 270 ENDPROC
 280 *
 290 *
 300 *
 310 DEF FNmodx(x,xx)
 320 =x+xx
 330 DEF FNmody(y,yy)
 340 =y+yy
 350 *
```

```
  360 *
  370 *
  380 DEF PROCball1
  390 LOCAL x,y,z
  400 LET x = x1: LET y = y1: LET z = 1
  410 COLOUR 0: PROCplot(x1,y1,CHR$ 224): LET x1 = FNmodx(x1
,xx1): LET y1 = FNmody(y1,yy1)
  420 IF x1<2 OR x1>18 THEN LET xx1 = - xx1: PROCsound
  430 IF y1<2 OR y1>28 THEN LET yy1 = - yy1: PROCsound
  440 PROCplot(x,y," ")
  450 ENDPROC
  460 *
  470 *
  480 *
  490 DEF PROCball2
  500 LOCAL x,y,z
  510 LET x = x2: LET y = y2: LET z = 2
  520 COLOUR 2: PROCplot(x2,y2,CHR$ 224): LET x2 = FNmodx(x2
,xx2): LET y2 = FNmody(y2,yy2)
  530 IF x2<2 OR x2>17 THEN LET xx2 = - xx2: PROCsound
  540 IF y2<2 OR y2>28 THEN LET yy2 = - yy2: PROCsound
  550 PROCplot(x,y," ")
  560 ENDPROC
  570 *
  580 *
  590 *
  600 DEF PROCball3
  610 LOCAL x,y,z
  620 LET x = x3: LET y = y3: LET z = 3
  630 COLOUR 3: PROCplot(x3,y3,CHR$ 224): LET x3 = FNmodx(x3
,xx3): LET y3 = FNmody(y3,yy3)
  640 IF x3<2 OR x3>17 THEN LET xx3 = - xx3: PROCsound
  650 IF y3<3 OR y3>28 THEN LET yy3 = - yy3: PROCsound
  660 PROCplot(x,y," ")
  670 ENDPROC
  680 *
  690 *
  700 *
  710 DEF PROCsound
  720 SOUND z,-15,z*60,1
  730 ENDPROC
```

PROJECTS

1. There's a BUG in Programs 16 and 17 that causes the program to stop if a$ (line 40) is " " (in other words, the user presses RETURN alone). Modify Program 17 (and if you're keen 16 as well) to prevent this from happening. You will need to think a little more about strings for this.
2. Write a BASIC program to fill the screen with the user's name in the colour of his/her choice on a background of chosen colour. Write the program twice—once using GOSUBs and once using PROCs; compare the results.
3. Study the programs listed at the end of the book which use GOSUBs or PROCs. Make sure you understand how and why they are used. Draw a memory map with control flow lines (like Figure 13.4 on Page 77) for one of the simpler ones.
4. You can animate pictures like this, a more simple example than Program 18.

```
10 VDU 23 225 etc
50 LET a$ = "    "+CHR$ 225+CHR$ 226+CHR$ 227
60 REPEAT: FOR place = 0 TO 13 in Mode 5
70      PROCprint
80      NEXT place: pause
90    CLS: UNTIL FALSE
```

```
 990 *
1000 DEF PROCprint
1010 PRINT TAB(place,15) a$
1020 ENDPROC
```

Try this simple example (and then study Program 18 and Program A2) and attempt more sophisticated routines of your own.

14 Topping Out Ceremony

I've been leading up to this for a long time—a few pages on program design. The ideas here become more and more important the longer your program. I don't think you can afford to ignore them (unless you have lots of spare time and enjoy frustration) once your programs start to pass the 30-line mark.

Not even the world's best programmer will be able to write a program of more than 20 or 30 lines without some sort of plan, even if only in his/her head. So it's time for us—*not* the world's best—to talk in detail about planning.

The time has come, because you now have met all the standard structural facilities of the Electron: REPEAT . . . UNTIL and FOR . . . NEXT loops, IF . . . THEN . . . ELSE, and PROCs and GOSUBs.

PROGRAM PLANNING

This involves two concepts that I've already introduced—the concepts of flowcharts and modules. I hope you'll recall that a *flowchart* is a diagram showing how the different parts of a program relate, and each of those parts is a *module*, a section with a single clear function. A third concept we've met is the *subroutine*—a program section that corresponds to a flowchart box. Subroutines can be open—in the main routine—or closed (in which case we here include procedures).

So here are the stages of development of a program, set out using those words.

Stage 1 Have an idea . . .
THEN break it down into concepts—modules.
Stage 2 Design the outline flowchart . . .
Each box relates to a module.
Stage 3 Write and test the program . . .
Each subroutine relates to a flowchart box.

The alternative looks much simpler:

Stage 1 Have an idea.
Stage 2 Write the program.

Please don't be tempted by that! Programming is not like writing a letter to a friend—the result must be *very* precise in the way it carries out its task. In a letter to a friend you can get away with things like 'Oh, by the way' and 'I forgot to say that' and 'PS'—in a program you can't. (Well, you *can* in a way, but truly I don't advise it.)

Let me show you what I mean, by developing a simple program in these three stages. It *has* to be a simple program, because I'm short of space—so it *has* to be rather trivial. Still that'll make it easier for me to show what I'm talking about . . .

IDEA TO MODULES

This is Stage 1—have an idea and break it down into modules.

Idea 'I'd like a program that prints two input numbers in order and so gives the mean (average).'

That statement already has some obvious concepts in it:

Concepts 1
(a) Accept two numbers.
(b) Output them in order.
(c) Output their mean.

That would be quite enough if we were giving instructions to a person—but micros aren't so clever, so we must be more precise:

Concepts 2
(a) Ask for two numbers.
(b) Accept two numbers; reject other inputs and re-start.
(c) If the numbers aren't in order, put them in order.
(d) Output the two numbers.
(e) Work out the mean.
(f) Output it.

That's much more precise now. But in practice we want a program to keep restarting until told to stop. That's because we're thinking of a user who has a whole set of pairs of numbers to process in this way. (It would hardly be worth writing a program for use only once.) And in practice, our program must help the user as much as possible. (It must be '*user-friendly*'—the user must never be in any doubt what to do and must never feel threatened.)

The restart-until-told-to-stop is best handled with what folk call a 'rogue value'. When the micro comes across this it knows it needn't go on. Let's use the letter 'S' for this purpose. Taking the rogue value and the user-friendly thing gives us this final list of concepts. I'll now call them modules.

Idea A program to print input pairs of numbers in order and give their mean.

Modules
(a) Say what the program does.
(b) Ask for two numbers.
(c) Accept two characters.
(d) If input is "S", stop with suitable message.
(e) Reject (politely) non-numeric inputs and return to (b).
(f) If the numbers aren't in order put them in order.
(g) Output the two numbers with suitable message.
(h) Work out the mean.
(i) Output mean with suitable message.
(j) Return to (b) for next go.

MODULES TO FLOWCHART BOXES

Stage 2 is very easy—we just have to express the list of modules as a picture—a flowchart, in which each module has led to a box. This should show clearly the exact lines of flow through the program. Try to do one for this program before looking at my version in Figure 14.1.

Drawing up the flowchart, and coding the program, may show the need for other modules in practice. Program development is often much less precise than my three broad stages imply!

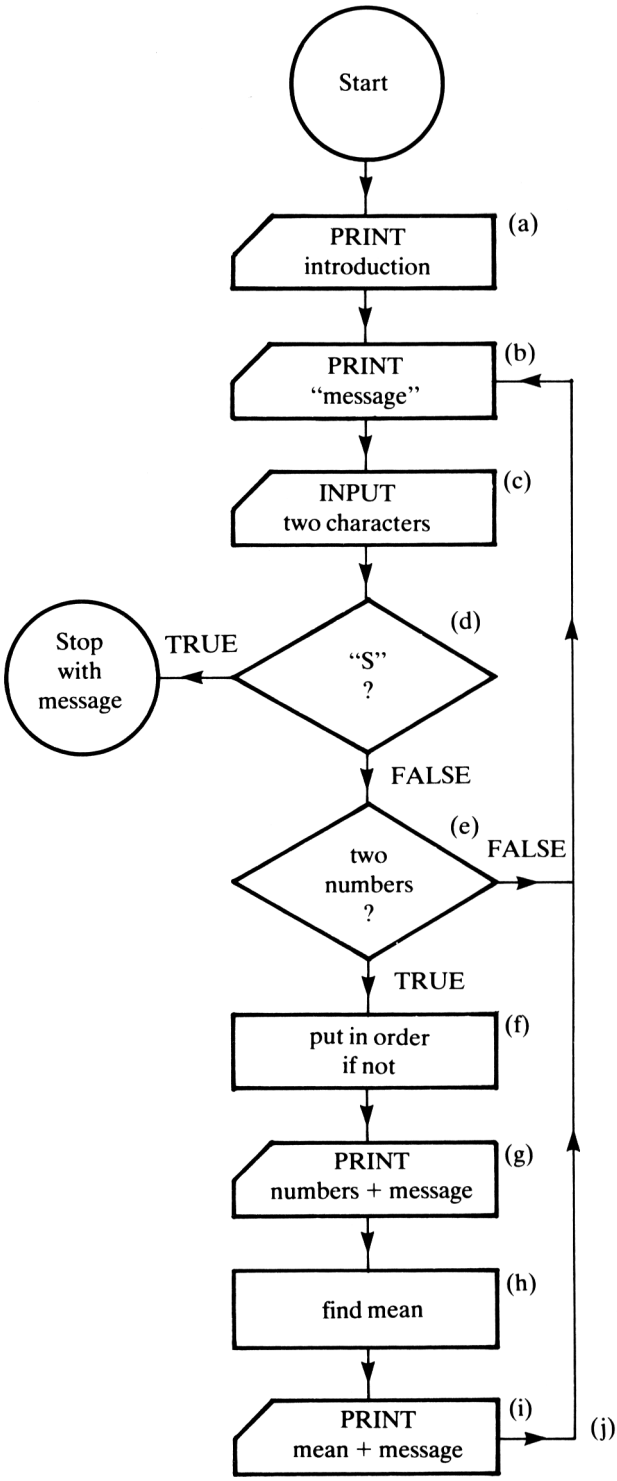

Start
PRINT
introduction
(a)
PRINT
"message"
(b)
INPUT
two characters
(c)
Stop
with
message
TRUE
"S"
?
(d)
FALSE
two
numbers
?
(e)
FALSE
TRUE
put in order
if not
(f)
PRINT
numbers + message
(g)
find mean
(h)
PRINT
mean + message
(i)
(j)

Figure 14.1

CODING

Writing and testing the program itself (Stage 3) should now be (almost) a doddle. The only worries are how to code each stage, for we expect the stages to relate to each other correctly.

All we need to do is to write the program lines for each module/flowchart box, either as an open subroutine (part of the main program) or as a closed subroutine (a separate chunk). I shall tend to the former, and explain why later. We can test as we go along.

Before we start coding we need to lay out a 'memory map' of the program. This can often be fairly complex, with several separate blocks of closed subroutines. In our case that's not needed so our map is simple, as shown in Figure 14.2. The main program starts at line 100, after any "initialization" (= start up) lines we may decide to include. The subroutines start at 10000.

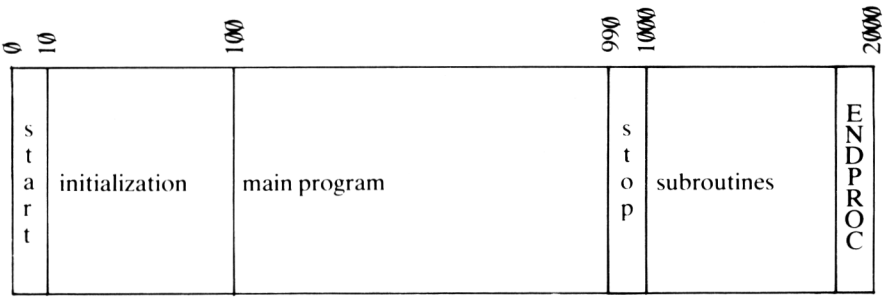


Figure 14.2

Here's our starting program skeleton then:

Program 19: Numbers (under construction)

```
    9 REM  **  Initialisation
   10 MODE 1: COLOUR 130: COLOUR 0: CLS
   11 REM  *   MODE 1 has 40 characters on a line - easier t
o read; easier to format
   99 REM  **  Main program
 1990 ENDPROC
 1991 REM  *   Lest we forget
```

Note

In the following program fragments, two-star REMs are part of the program itself, while the measly one-star REMs are extra passing comments.

Of course do set up

*KEY 10 "OLD|M|N LIST|M"

to set up a paged listing whenever you use BREAK.

Module (a) Say what program does.

```
  100 PRINT' "PROGRAM 18"' "==========="
  110 PRINT''' "This program will put your two input    numb
ers in order, and also give their   mean (average)."
```

Module (b) Ask for two numbers.

```
  120 PRINT''' "   Please type in two numbers, or S to  stop
."''
  130 PRINT "   Press RETURN after each answer."
```

Module (c) Accept two characters.

```
  140 INPUT char1$, char2$
 1100 DEFPROCwait
 1110 FOR wait = 1 TO 1000:  NEXT wait
 1120 ENDPROC
```

Module (d) If (either) input is "S" stop with suitable message.

```
  150 LET flag = 0: IF char1$ = "S" OR  char1$ = "s" OR char
2$ = "S" OR char2$ = "s" THEN PROCstop: IF flag = 0 THEN END
 ELSE IF flag = 1 THEN COLOUR 130: CLS: GOTO 120
```

In line 150 I introduced a variable called 'flag'. I use the name on purpose—by the word 'flag' computer folk mean a variable whose value is 0 or 1 depending whether something is false or true. So the value of 'flag' here is going to tell us whether the user wants to stop (flag = 0) or not (flag = 1). The value of 'flag' is set in line 150, changed if need be in line 1060, and checked on return from that procedure in the end of line 150.

Module (e) Reject (politely) other inputs and return to (b).

```
  160 LET flag = 0: PROCcheck
  990 END
  991 REM  *   Protect prog from subs
  999 REM  **  Subroutines
 1000 DEF PROCstop
 1001 REM  *   The first closed subroutine
 1010 LOCAL resp$
 1020 COLOUR 129: CLS
 1030 PRINT'' "You entered 'S' - do you want to stop?"'' "Pl
ease type Y(es) or N(o) & RETURN."'''
 1040 INPUT resp$
 1050 IF resp$ = "Y" OR resp$ = "y" THEN PRINT''' "Thank-you
!"''' "To start again, type RUN & RETURN....": ENDPROC
 1060 IF resp$ = "N" OR resp$ = "n" THEN PRINT'' "OK - we'l
l start again!": PROCwait: LET flag = 1: ENDPROC
 1069 REM  *   Here only if neither Y nor N
 1070 GOTO1040
 1071 REM  *   Two ENDPROCs, neither at end of PROC!
 1200 DEFPROCcheck
 1210 LET c1 = VAL char1$: LET c2 = VAL char2$
 1211 REM  *   See text below
 1220 IF c1 = 0 AND char1$ <> "0" THEN LET flag = 1
 1230 IF c2 = 0 AND char2$ <> "0" THEN LET flag = 1
 1240 ENDPROC
  170 IF flag = 1 THEN PROCinv: CLS: GOTO 120
 1500 DEFPROCinv
 1510 COLOUR 129
 1520 PRINT SPC(40) "These aren't valid numbers:" SPC(63) "P
lease re-enter!                "'''
 1521 REM  *   Study the effect of SPC & why I use it here!
 1530 PROCwait
 1540 COLOUR 130
 1550 ENDPROC
 1551 REM  *   So next line no longer needed
```

See another flag in use in this section—to show whether the input characters are numbers (flag = 0) or not (flag = 1). There's no harm in using the same variable name—the routines don't overlap at all.

I said 'see below' by line 1210, because that line needs comment. It introduces a new *string function*, called VAL. What VAL does is to turn the string argument (any collection of any kind of characters, you recall) into a number. If the string contains any

characters other than number ones (only full stop and E allowed apart from the digits), the result of VAL is 0.

So in line 1210 LET c1 = VAL char1$ tries to turn 'char1$' into a number. If it fails 'c1' gets to be 0. However 'c1' will also get to be 0 if char1$ is '0'—that's why the next line has that AND in it. The same applies to c2 and char2$ in lines 1210 and 1230.

Module (f) If the numbers aren't in order, put them in order.

Module (g) Print the two numbers with suitable message.

As Module (*f*) is so easy with only two numbers, I put those two chunks together.

```
 180 PROCorder
1300 DEFPROCorder
1310 PRINT''' "Your two numbers in order are:"''
1320 IF c1 < c2 THEN PRINT c1, c2 ELSE PRINT c2, c1
1330 ENDPROC
```

Module (h) Work out the mean.

Module (i) Print mean with suitable message.
Home straight now—and again we can combine:

```
190 PRINT''' "Their mean is "; (c1+c2)/2 "."
```

Module (j) Restart.

```
 200 PROCwait: PROCwait: COLOUR 131: PRINT TAB(2,25) "Pleas
e press RETURN to re-start....": COLOUR 130: PRINT''''
 210 LET r = GET: CLS: GOTO 120
 211 REM  *   So next line is no longer needed
```

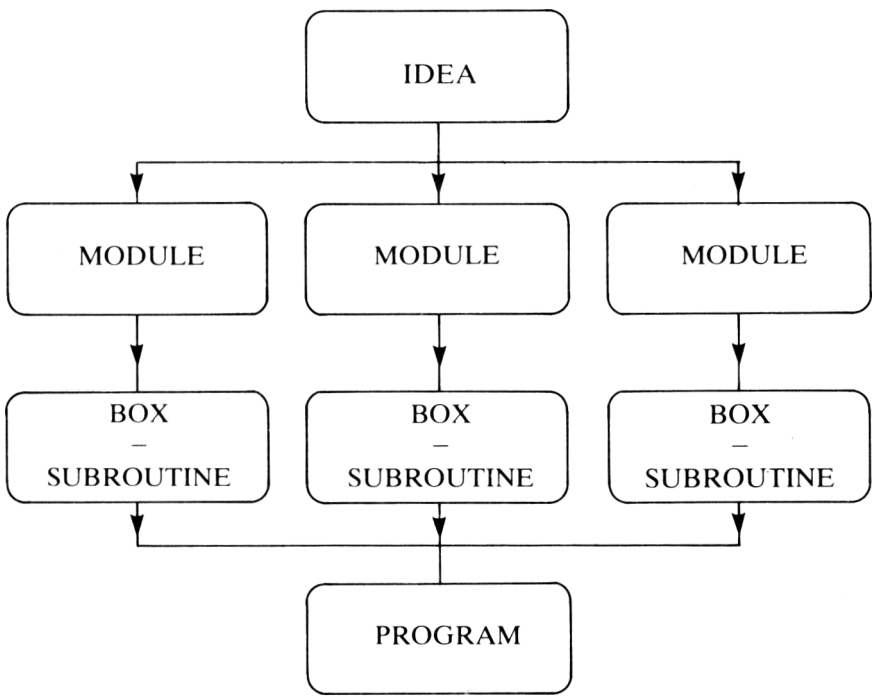


Figure 14.3

Note a couple of useful tricks there. That new bit using GET in line 210 is a snazzy trick to cause the computer to stop and wait until a key is pressed. Very useful indeed! (There will be more on this in the next chapter.) Then, slipping back a line, note the use of COLOUR to highlight special text. I've done the same in other parts of the program, such as PROCinv (from line 1500). Mode 1 is very good indeed for producing colourful text readily—you get 40 characters on a line, and can play with 4 colours on the screen at once.

That's now a good implementation of the original idea (even if the original idea wasn't such a good one). It can be improved, I'm sure, and it can easily be made more complex. But it appears here as a demo of developing programs. Remember? Remember!

The posh name for this kind of program development is modular, or *top-down programming*, hence the title of this chapter. I can easily sketch the approach like the diagram in Figure 14.3; then the reason for the name is clear.

FINISHING

So that's our three stages. But now I'll be honest and admit there's a fourth. I can't *really* call it finishing, as Deeson's First Law of Programming is that *no program is ever perfect*. Polishing is a better word. This includes exhaustive tests and 'debugging', because even good top-down programming cannot avoid every error. (I don't really call myself a good top-down programmer, especially when I am trying to use more simple program structures than I normally would; and I must admit that I spent almost an hour trying to find a bug in one of the lines of the above program!)

We'll look at debugging in a couple of chapters. But polishing also includes maximizing *user-friendliness* (good messages, nice layout, proper grammar and so on, effective use of colour and sound), and it includes an ongoing process of incorporating new twiddly bits like more effective colour/sound/graphics, better instructions, and so on.

Mind you, while I believe it's true that no program is ever perfect, there is a danger of overpolishing a program just as with overpolishing a floor—it can easily cause you to slip up and you then lose the wood of the central idea in the trees of twiddly bits. Some programs are like a mediaeval illuminated manuscript—beautiful but not of great practical value.

STRUCTURED PROGRAMMING

The top-down modular development we've been thinking about in this chapter is part of what is called *structured programming*. This refers to a sign of efficiency, to which the Beebasic of the Electron lends itself fairly well. It makes program listings crystal clear to read and pretty efficient to the computer. When done properly that is. It uses procedures and REPEAT . . . UNTIL structures, and a minimum of references to line numbers (i.e., a minimum of GOTOs and GOSUBs in the case of the Electron).

The programs in this book are not as highly structured as some people would like, as it is my view that such an approach would be unfair to you. All the same it is crucial that you consider this if you intend to become a classy programmer, and I recommend particularly your working through the book *Structured programming with the BBC microcomputer* by Roy Atherton (Ellis Horwood, 1982).

In fully structured form the program developed in this chapter would start off rather like Program 18 in the last chapter, but each successive DEF PROC would include further PROCcalls. As I said when discussing *that* program, such a procedure (ho, ho) can become ridiculous.

The essence of structured programming is that the program listing does not require line numbers at all. However, with the Electron, you *do* have to put these in, even if they aren't necessary. If you scan through Program 19 (if you can find it amongst all that text), you will see that line numbers aren't needed.

PROJECTS

1. Redesign Program 19 to work on three numbers. That does not in fact require many changes, but one particularly important one would be that if the first input character is "S", the program should go over to the STOP routine at once.
2. Develop from scratch an interactive number-processing program, perhaps a bit like Program 19, using the principles discussed in this chapter and previously.

15 String Them Up

It's just possible—though I hope it's *not* the case—that you may be feeling that what I've done so far is all somewhat mathematical. I'm afraid that if you do think that, there's not much that can be done about it—computer programming *is* rather mathematical. By that I mean that it has to use numbers and relationships and has to involve fairly careful logical thinking.

The following direct command (direct so it doesn't need a line number), I agree, may not appear to be mathematical.

```
PRINT "Viva el Elk."
```

All the same, it really is mathematical in that:

1. Each character has to be converted to a number in the computer.
2. The numbers that result are grouped following certain rules—mathematical ones if you like.
3. The numbers move around in the micro following more such rules.
4. Mathematics, in a sense, is the process involved when the command is carried out, or when it is rejected because of an error.

All the same we don't need to think about all that, and in this chapter I'm going to tell you about *strings*—and we don't need to be at all mathematical here. Logical, yes, though!

WHAT IS A STRING?

As you've picked up long ago, a string is *any* set of characters. The characters in it can be alphabetical letters, numbers, punctuation marks, symbols—anything you can get from the keyboard. So "Viva el Elk." is a string. So is "I have a 32K memory.", and so are "3 × 6 = 18", "The year is/was 1984.", "What is your name?", "+,(-./<6a *Z", "24", and "□ □ □".

All these are what we call *string constants*—that means they don't change. String constants are enclosed by quote marks (". . .") in writing and when communicating with the computer.

(There are two interesting exceptions to the last point—string constants do not need quotes when entered by an INPUT statement, and they are not always required when held in DATA lines. I'll come back to the latter point later, but we have already discussed the former.) This quote mark rule means that *strings cannot contain quote marks*. Try this, for instance—and see why it can't work:

```
PRINT "I am the "Electron" computer."
```

The computer manages the first part—treating "I am the" as a string constant and printing it. Then it's told to PRINT Electron. It can do that only if Electron is a numeric variable with a value, say, 2Ø. Here it isn't, so we get the error report "No such variable".

If you need to use quotes in a string use the apostrophe:

PRINT "I am the 'Electron' computer." will be OK.

(There are complex ways to get double quotes into a string constant, but why bother?)

As well as having numeric constants (like 4) we have numeric variables (like "numb", where "numb" could be *any* number). In the same way, again as you already know, we can have *string variables*. Here's one—N$ (say it "N string"). The dollar symbol after a variable name shows that this is a string variable. We've seen how to use this in earlier programs.

It's like this:

```
10 CLS
20 INPUT' "What is your name, Player 1", name1$
30 PRINT'' "Thank-you, " name1$ "."
40 INPUT''' "And what is yours, Player 2", name2$
50 PRINT'' "Thank-you, too, " name2$ "."
60 FOR wait = 1 TO 1000: NEXT wait
70 CLS: PRINT' "Right, " name1$ " and " name2$ ", what yo
u do is..."
80 and so on
```

Each time the program runs, the values of name1$ and name2$ could change—they are variables.

Electron BASIC has the following rules for "allowed" string variable names. The string variable name can be one or more characters. The first must be a letter (or £); the others can be only letters and/or numbers (or £ or _); the thing must end with $ of course. The length can be as much as you like (well, up to about 230 characters)—but as usual long names need care. Note that the letter(s) in a variable name can be upper case (capitals) or lower case (CAPS LK light off). The former are easier but most folk prefer lower case. This is because *a variable name should not contain a keyword* (like PRINT or TO).

As with numeric variable names, then:

1. You need a different name for each variable, unless there's no chance of overlaps.
2. It's nice for a name to tell the reader something of its meaning (for instance, name$), but:
3. Long names take up a lot of memory and can too easily be mis-typed.

Also Electron BASIC has the same rules for using string constants and variables in PRINT statements as for numeric ones. Beware the semi-colon (;) though (even if omitted)—you need to watch the spacing round about. See how I structured and spaced the string variable and constants in line 70 above, for example.

STRING OPERATORS

We can do all sorts of things with numeric constants and variables. We can add, subtract, raise to a power, multiply, and divide, and we can use functions like SIN and SQR. Well, we can add strings too, but the other operations do not have much meaning and can't be carried out as such.

The posh name for adding strings is *concatenation*. This means chaining together, and we use the + sign as with numbers. I've used the technique already, without special comment. See, for instance, Program 16; here's another example.

```
10 INPUT "What's your first name", first$
20 INPUT "And your second", second$: REM Now you know what
the phrase 'second string' means!
30 INPUT "Are you male (M) or female (F)", sex$
40 CLS: PRINT' "Thank-you, ";
```

```
  50 IF sex$ = "M" OR sex$ = "m" THEN PRINT "Mr "; ELSE PRINT
"Ms ";
  60 LET name$ = first$+" "+second$
  70 PRINT name$; "."
```

The concatenation bit is in line 60. Got it?

In a PRINT statement we can have (no spaces required)

either	PRINT x$ y$	[= P. x$;y$]
or	PRINT x$ + y$	

Both have the same effect, but the latter (with +) is in fact a little smoother in action.

The above program also showed that we can use *logical operators* with strings, in decision statements—line 50. In fact we can use all the logical operators here: <, < =, =, > =, and >, as well as AND, OR and NOT. Thus we can compare strings.

But what does that mean—how can one string be 'greater' or 'less' than a second? The ordering is in fact based on *ASCII codes*. Each character has a code, and what the micro does is to compare the strings character code by character code. (It has to do the same with the strings used in variable names and as that takes time it's another reason for using short names.)

Here are the main ASCII codes in summary.

Codes	Characters
0–31	(Special effects—see *User Guide*, Page 105)
32–47	Symbols
48–57	Numerals
58–64	Symbols
65–90	A–Z
91–96	Symbols
97–122	a–z
223–250	User defined (Chapter 5)

And here's a full list of the main ones, codes 32–126.

Code	Character	Code	Character
32	□ (space)	59	;
33	!	60	<
34	"	61	=
35	#	62	>
36	$	63	?
37	%	64	@
38	&	65–90	A–Z
39	'	91	[
40	(	92	\
41	)	93	]
42	*	94	^
43	+	95	_
44	,	96	£
45	-	97–122	a–z
46	.	123	{
47	/	124	|
48–57	0–9	125	}
58	:	126	~

Using these codes the micro can order strings in the same way as we can put names in alphabetical order. Thus:

A SMITH	<	B SMITH
SMITHA	<	SMITHWH
SMITH	<	SMITHSON
SMITH1	<	SMITH2
12345	<	54321
12345	<	SMITH
SMITH	<	Smith
$MITH	<	SMITH

and so on. Please check that you agree with the above using the ASCII code list!

The next program uses all this to sort a list of input strings into alpha (or ASCII) order. The input strings are called in$(1), in$(2) and so on up to in$(limit). This technique uses what's called a *string array* (= list of strings). You don't need to know more about it at the moment than this: before using an array tell the micro how big it is with a DIM (= dimension) statement. That's why I've put DIM in$(limit) near the beginning of the program—it tells the computer I'm going to have 'limit' strings called in$(1) . . . in$(limit), and then the micro goes off and reserves space for them.

The important thing here about this program is the sorting routine. Each time the routine is reached, the micro puts the two strings in question into the right order IF they need it. Spend a while working out how that routine sorts—it uses a valuable idea, that of a temporary store (here called s$).

Program 20: Alphasort

```
  10 PROCsetup
  20 PROCsetlimit
  30 PROCinput
  40 PROCsort
  50 PROCoutput
  60 REPEAT: UNTIL FALSE
  61 REM  **  End of main routine
  62 REM  **  ESCAPE to escape
  70 DEF PROCsetup
  79 REM  **  NB simpler layout than before
  80 VDU 19 128 4 0;0
  81 REM  **  Sets paper to blue
  90 VDU 19 1 3 0;0
  91 REM  **  Sets ink to yaller
 100 CLS
 110 ENDPROC
 120 DEF PROCsetlimit
 130 INPUT' "How many names to sort", limit
 140 DIM in$(limit)
 141 REM  **  There's that DIM thing
 150 ENDPROC
 160 DEF PROCinput
 170 FOR in = 1 TO limit
 180   PRINT'' "Enter name "; in ": ";: INPUT SPC(5) in$(in
)
 181   REM  **  SPC again
 190   NEXT in: LET in = in-1
 191 REM  **  Why LET in = in-1?
 200 ENDPROC
 210 DEF PROCsort
 220 FOR a = 1 TO in-1: LET flag = 0
 230   FOR b = 1 TO in-a
 239     REM  **  Gulp....
 240     IF in$(b)>in$(b+1) THEN LET s$=in$(b): LET in$(b)=
in$(b+1): LET in$(b+1)=s$: LET flag = 1
 241     REM  **  There, that's the sorting line
 250     NEXT b
 260   IF flag = 0 THEN ENDPROC
```

```
270   NEXT a
280 DEF PROCoutput
290 VDU 14
291 REM  ** Program equivalent to CTRL+N
300 CLS: FOR a = 1 TO in
310   PRINT SPC(10) in$(a)
320   NEXT a
330 ENDPROC
```

If you can't understand the rest of this program, don't worry—it's a very efficient one, but that makes it hard to grasp. Just concentrate on the string-handling in line 240.

So, in a sense we *can* do 'arithmetic' with strings, either constants or variables:

1. We can add strings, like this
   LET slogan$ = name$ + " rules OK!"
2. We can compare strings, like this
   IF a$ < b$ OR c$ > = "TEST STRING" THEN LET d$ = e$
3. And we've used STRING$, as in LET line$ = STRING$ (32, "="), which is a bit like multiplying. See below for more of this.

STRING FUNCTIONS

In Chapter 9 we looked at various numeric functions. Things like INT and SQR, these act on numeric (constant or variable) arguments to produce numbers.

Thus INT (3.6) gives 3 and SQR (num) gives 2 if num = 4.

BASIC also has various string functions. We've already met one—LEN, which gives the number of characters in the string argument.

LEN "Good morning!" = 13
and LEN name$ = 5 if name$ = "Acorn"

LEN is a string function that produces a numeric result. Some string functions produce a string result. STRING$ is one I find useful; it's the nearest we've got (on any computer I know) to multiplying strings. We've met it before, but it is nice and fairly important, so let's have a little more about it.

STRING$(n, x$) produces a string of n copies of x$ added together. Thus LET b$ = STRING$(2, "bye") puts "byebye" into b$. Again LET blank$ = STRING$(40, "□") makes blank$ into a line of forty spaces.

Here are some more string functions. I'll give a sample program using them after the notes!

ASC ". . ." comes first. It gives the ASCII code (page 94) of the first character in the string that is its argument. PRINT ASC "Acorn" gives 65 because 65 is the ASCII code of 'A'. The opposite of ASC is:

CHR$ n which gives the character whose ASCII code is 'n'. The value of 'n' should be between 0 and 255. (Well, if it isn't, you get what posh folk call a 'wrap-around' effect, giving just a reflection of what *is* in the range 0–255). Some n's give useful effects rather than characters. Thus PRINT CHR$ 12 will clear the screen, while CHR$ 8 has the effect of 'back-space'. Try—PRINT "I am the □" CHR$ 34 "Electron" CHR$ 34 "□ computer". That should neatly solve a problem we met a little earlier.

Note

VDU n = P. CHR$ n. That's of value for direct commands and in programs outside PRINT statements. Thus VDU 12 is equivalent to CLS; VDU 7 buzzes the speaker. (Perhaps you should note that strictly VDU n = P. CHR$ n;.)

GET$ waits for a key press and then gives a string containing the character pressed. For instance,

```
100 PRINT "Press 1, 2, 3 or 4."
110 REPEAT: LET choice$ = GET$
120 UNTIL choice$ > "0" AND choice$ < "5"
```

So it's a bit like INPUT but (a) doesn't give a prompt, (b) doesn't need (R), and (c) accepts only one character. There is a similar useful numeric function, GET, which is just the same except that it gives the ASCII value of the character entered.

*INKEY$* (and its numeric counterpart INKEY) is almost the same as GET$, but this time it waits for a keypress only for a certain time. The time to wait is given as a number argument and works in hundredths of a second. Thus the previous example could be like this:

```
100 PRESS "Press 1, 2, 3 or 4."
110 LET choice$ = INKEY$ 200          [two seconds' wait]
120 IF choice$ = " " [i.e. nothing] THEN PRINT "Too late; my turn now."
130 IF choice$ < "1" OR choice$ > "4" THEN PRINT "Mistake;
    my turn."
```

INSTR(a$, b$) reports where in a$ the micro first finds b$. If the search is not successful the result is 0. Of course a$ must be longer than b$ for there to be any hope at all of success.
Here's an example:

```
100 INPUT "What's the capital of Argentina", ans$
110 IF ans$ = "BUENOS AIRES" THEN PRINT "Right!" ELSE IF
    INSTR(ans$, "AIR") THEN PRINT "Nearly—BUENOS AIRES" ELSE
    PRINT "No—it's BUENOS AIRES."
```

Note: Never use a space after INSTR; always put brackets in.

LEFT$(big$, 3) gives the first three (left-most three) characters of big$. Note there mustn't be a space after LEFT$. Thus:

```
100 INPUT "Name", name$
110 IF LEN name$ > 6 THEN PRINT "That's too long for me to remember.
    I'll just call you □" left$ (name$, 6): LET name$ = LEFT$ (name$, 6)
```

Note that this won't cause any problems if LEN name$ is less than or equal to 6.

MID$(big$, 3, 2) is rather similar—it gives the third and fourth characters of big$: the middle, starting at character 3 and taking 2 characters. Thus MID$("The Electron computer", 5, 8) = "Electron", and you can use things like

```
IF MID$(a$, 9, 12) = LEFT$(b$, 4) THEN . . . or LET d$ = MID$(c$, 7, 5)
```

And then, of course, there's:

*RIGHT$* which is just the reverse of LEFT$, counting from the righthand end of the string concerned instead of the left. So RIGHT$("The year is 1984.", 5) = "1984."

*STR$* sounds strange, but is really very useful. It turns the number that follows it into a string. Then we can use it as a string instead of a number. So STR$ 1984 = "1984". If, for instance, you want to print a long variable number, "bignum" at the centre of line 20 in MODE 1 (which has 40 characters on a screen line), use this:

PRINT TAB((20 − LEN STR$ bignum) / 2, 20); bignum.

Can you work that out? It's a valuable trick.
The opposite of STR$ is:

VAL which turns a string into a number (if the string contents are numeric). Thus VAL "1984" = 1984, but things like VAL "Electron" give 0. One major use of this is to check that a numeric input *is* a number and not a mistake. Like this:

```
100 INPUT "Give me a number.'' '' ans$        [useful layout trick here!]
110 IF ans$ <> "0" AND VAL ans$ = 0 THEN PRINT "I said—a
    NUMBER!!": GOTO 100
```

Phew—there are quite a lot of string functions, aren't there? They all have major uses. Only practice can make you really familiar with them. Before I give you a little of that practice, there's one more paragraph.

It's to note that, as with numbers, you can define your own string functions. The rules are the same (see Chapter 9):

1. Give the function a name, including variable names concerned in brackets. Let's call it rep(A$).
2. Define it somewhere in the program with DEF FN . . . = . . . Thus

   100 DEF FNrep(A$) = STRING$ (n, A$)

3. Call it when needed with FNname(variable), like this:

```
110 LET n = 5: LET g$ = "Hi!": PRINT FNrep(g$)
150 LET n = 3: LET p$ = FNrep(ans$)
190 PRINT FNrep("□ THE END □")
```

I doubt you'll have much need for defining your own string functions yet awhile, but that's enough to start you off when you do. Anyway, now I'll have a go at a program to use strings and string functions.

Program 21: Cat's cradle

(Laugh—that's a joke about strings; we'll come to a real cat's cradle later . . .)

```
 10 MODE 5: PROCsetup
 11 REM  *   No MODE in PROCs
 20 PROCcradle
 30 PROCname
 40 PROCdate
 50 PROCfinal
 51 REM  *   End main routine; final doesn't end
100 DEF PROCsetup
110 COLOUR 129: COLOUR 0
120 PRINT CHR$ 12 CHR$ 10 "Ready when you are. ..."
121 REM  *    = CLS: P.'
130 LET go$ = INKEY$ 100
131 REM  *   Wait 1s
140 IF go$ = "" THEN PRINT CHR$ 10 CHR$ 10 "Hurry up...."
150 LET go$ = GET$
151 REM  *   Wait for ever
160 VDU 19 129 2 0;0: PRINT CHR$ 12 CHR$ 10 CHR$ 10 "About
time too!!"''''
161 REM  *   Change paper to green; CLS, P.''
```

```
  170 ENDPROC
  200 DEF PROCcradle
  210 VDU 19 0 4 0;0: LET a$ = "Cat's cradle!"
  220 LET a = 0
  230 REPEAT
  240   LET a = a+1: PRINT TAB(RND(20-LEN a$)) LEFT$(a$,RND(
LEN a$)): PROCwait(50)
  250   UNTIL a = 100
  260 VDU 19 129 3 0;0
  270 LET a = 0
  280 REPEAT
  290   LET a = a+1: PRINT TAB(RND(20-LEN a$)) MID$(a$,RND(L
EN a$),RND(LEN a$)): PROCwait(50)
  300   UNTIL a = 100
  310 VDU 19 0 7 0;0
  320 LET a = 0
  330 REPEAT
  340   LET a = a+1: PRINT TAB(RND(20-LEN a$)) RIGHT$(a$,RND
(LEN a$)): PROCwait(50)
  350   UNTIL a = 100
  360 LET a = 0
  370 REPEAT
  380   LET a = a+1: PROCopen(a$): PROCwait(50)
  390   UNTIL a = 20
  400 VDU 19 129 0 0;0: PRINT''''''"RETURN!"
  410 LET g = GET
  420 ENDPROC
  450 DEF PROCwait(lmt)
  460 FOR w = 1 TO lmt: NEXT w
  470 ENDPROC
  500 DEF PROCopen(o$)
  510 LOCAL a
  520 IF INT(LEN o$/2) = LEN o$/2 THEN LET o$ = o$+"*"
  521 REM  *   Tests if o$ has even number of chaacters
  530 FOR a = 1 TO LEN o$*2-1 STEP 2
  540   LET b$ = MID$(o$,LEN o$/2+1-a*0.5,a): PRINT TAB(10-L
EN b$/2) b$
  550   NEXT a
  560 ENDPROC
  600 DEF PROCname
  610 COLOUR 131: COLOUR 1: CLS
  620 PRINT CHR$ 12 CHR$ 10 CHR$ 9 "PHEW!!!": PROCwait(500)
  630 PRINT''' "What's your name,    anyway?"''''
  640 INPUT name$: PRINT''' "Oh."''': LET pause$ = INKEY$(15
0)
  650 CLS: VDU 19 131 3 0;0
  660 FOR a = 1 TO LEN name$*2: PROCopen(name$): NEXT a
  670 COLOUR 128: COLOUR 3: PRINT''''' "RETURN!"
  680 LET g = GET
  690 ENDPROC
  700 DEF PROCdate
  710 VDU 19 128 5 0;0: PRINT CHR$ 12 + CHR$ 10 + "Gimme you
r date of  birth - like 050466."''': INPUT date$
  720 CLS: LET d = VAL date$
  721 REM  *   $ to numero function
  730 FOR a = 1 TO 10: PRINT; a " x "; d " = "; a*d: NEXT a
  731 REM  *   Some books make a whole program out of that o
ne line....
  740 PROCwait(1000): FOR a = 1 TO 10: PROCopen(date$): NEXT
 a
  750 ENDPROC
  800 DEF PROCfinal
  810 VDU 19 128 13 0;0: LET final$ = STRING$(200/LEN name$
+ 1, name$+"*")
  820 REPEAT
  830   PRINT final$'''': PROCwait(150)
  840   UNTIL 0
  841 REM  *   Short for UNTIL FALSE
  850 ENDPROC
  851 REM  *   Not reaally needed!
```

To quote from line 62Ø—"PHEW!!!" Did I imply that string-handling is not too mathematical? Oh, well, it's fun anyway. On purpose I'm not going to dissect the above program further. That's up to you. There are some very useful routines in it—worth getting the hang of them! Turn now to the projects to reinforce the ideas before you forget.

PROJECTS

1. Study Program A10 at the back of the book in the light of string-handling.
2. In MODE 5, we can centre a set of characters (a$) in the line by PRINT TAB(1Ø − LEN a$ / 2) a$. If the set of characters is a number (a), use PRINT TAB(10 − LEN STR$ a / 2); a. Lay out a screen full of titles and so on (like the title page of a book) using this routine in a procedure.
3. Get the Elk's character set using a routine based on the following.

```
1Ø  FOR code = Ø TO 255
2Ø  PRINT code, CHR$ code
3Ø  NEXT code
```

   Beware the special effects of low values of "code" . . . Use VDU code to explore further.
4. Use as many string operations and functions as you can (and need!) in a program to:
   (a) accept items of personal data about the user;
   (b) display the complete set of data neatly on screen.
5. Develop Program 20 to your own purposes.
6. Devise a pattern-generating program using STRING$.
7. Explore GET, GET$, INKEY and INKEY$ as ways of getting, checking and using keyboard input. Add VAL if you wish.
8. Devise a program with INSTR used usefully and meaningfully.

16 Bug in a Rug

The story goes that in the very early days of electronic computing, far less than half a century ago, incorrect operation was often traced to the activities of insects making their homes among the nice warm circuits. Alas, users of modern computers no longer have that excuse for program failures. Still, the use of the word 'bug', to mean an unknown cause of software malfunction, remains. Here I would like to look very briefly at the process of 'debugging', one of the last stages of program development.

In Chapter 12 I tried to describe a systematic approach to program development—the top-down modular approach. I have also noted that a program written as a series of subroutines is much easier to test. That's because you can test each subroutine on its own. In this chapter I shall discuss debugging approaches with a complete bugged program rather than a subroutine.

A BUGGED PROGRAM

The following program contains three major faults and several minor ones. I'd like you to enter it; as you do this, by all means look out for the faults. If you find any—feel proud, but do not correct them!

Program 22a: Bugged bugs (initial version)

```
10 VDU 23 224 0,84,57,108,254,108,57,84
20 VDU 23 225 0,5,114,90,202,138,142,0
30 VDU 23 226 0,82,36,25,62,25,36,82
40 VDU 23 227 254,130,124,74,90,66,126,0
50 MODE 2: COLOUR 134: CLS
60 FOR bug = 1 TO 50
70   PRINT TAB(RND(20)-1, RND(30));
80   IF RND(4) = 1 THEN COLOUR 1: VDU 224: GOTO 120
81   REM  *   Recall that VDU n = P.CHR$ n
90   IF RND(4) = 2 THEN COLOUR 2: VDU 225: GOTO 120
100  IF RND(4) = 3 THEN COLOUR 3: VDU 226: GOTO 120
110  IF RND(4) = 4 THEN COLOUR 4: VDU 227: GOTO 120
120  FOR a = 1 TO 130 PRINT TAB(a) " BUGS ": NEXT a
```

Entered? Don't RUN for a moment. First—can you see what this program is set up to do?

Lines 10–40 define four bugs (small creature type) and 50 sets up a MODE 2 screen. Then, lines 60–110, we print fifty random bugs at random screen sites. After that, 120–130, we display a diagonal title and stop.

Not a fantastic program—but does it work? Try it. And you get one or two bugs—one

error message, and perhaps one bug of the VDU 23 type. The error report tells us of a syntax error in line 12Ø. Here's the line:

```
12Ø  FOR a = 1 TO 13Ø PRINT TAB a "BUGS": NEXT a
```

I hope you can find, and correct, the error yourself!
RUN—and a great mess of a diagonal title, with a screen bug (insect) first if you're lucky. No error report this time—the program has run happily to the end. But it hasn't worked as planned.

For a start—no fifty insects. And if we'd get any we'd lose them with that diagonal title. Ah, wait a moment—that should be 3Ø in line 12Ø, not 13Ø—a typing error, with a profound effect. 3Ø lines in MODE 2, not 13Ø! So line 12Ø should be:

```
12Ø  FOR a = 1 TO 3Ø: PRINT TAB a "BUGS": NEXT a
```

But it still doesn't work—the title doesn't start at the top of the screen, *and* it isn't a simple diagonal. The former is one of the three serious faults—can you explain it? Can you solve it? The latter is bad planning—there are thirty lines, but our title has six characters and there are only twenty character spaces in a line anyway.

To sort out both faults, *and* make the final cursor less annoying, several changes to 12Ø and 13Ø are needed. Here they are:

```
12Ø  PRINT TAB (Ø, Ø);: FOR a = Ø TO 14: PRINT ' TAB a "BUGS": NEXT a
13Ø  LET g = GET
```

Please try to understand these changes. The major fault is cured by the first statement in line 12Ø, but can you see the value of the other changes? But we still get no more than one "insect". Why's that? You've probably spotted the reason—major error number 2! The FOR in line 6Ø isn't matched by a NEXT anywhere. We *do* get an error report if there's a NEXT without a FOR—but the computer can't spot the reverse problem. (Why not?) So we have to watch out for it all by ourselves!

Well, we need a NEXT bug somewhere. And we need it before the closing credits, line 12Ø—so

```
115  NEXT bug
```

OK? Simple!
RUN. OK? *Not* simple!—there's no change in the display. What's wrong? I guess you can see it—the GOTOs in lines 8Ø–11Ø point at 12Ø rather than 115. We could change all those GOTO addresses, but let's be clever—delete 115, and put NEXT instead of GOTO 12Ø in each of those lines. That gives us a much neater listing, less wasted memory, and, at last, surely, a working program. RUN it.

Ah-ah. Still not fifty insects, are there? I tried to RUN this a couple of dozen times or more and the best I managed was nine. (Though some insects get 'squashed' by the title.)

So for some reason, the loop does loop but never gets anywhere near its official two score and ten goes. We can check that readily enough, using a valuable little debugging trick. Add at the front end of 7Ø—PRINT TAB (Ø, 1) bug:. Now each time you run, you can see how far 'bug' gets; nowhere near 5Ø, that's for sure. The printing out of carefully chosen variable values is an extremely useful aid to finding bugs in programs. (Don't forget to remove the statements concerned when they're finished with, though.) I expect by now you've found the value of editing methods . . .

TRACING

Electron BASIC has another useful debugging tool, called TRACE. TRACE ON causes the display of each line number carried out. If you use TRACE ON in a program, rather than as a direct command, you will need to use TRACE OFF to stop that display. In that case too you will find it helpful to have TRACE OFF fairly close in the program to TRACE ON, because otherwise you get a fairly meaningless screenful of line numbers.

What you *want* is to be able to trace the micro's path through no more than a dozen or so lines. You can then check on paper that the path is correct.

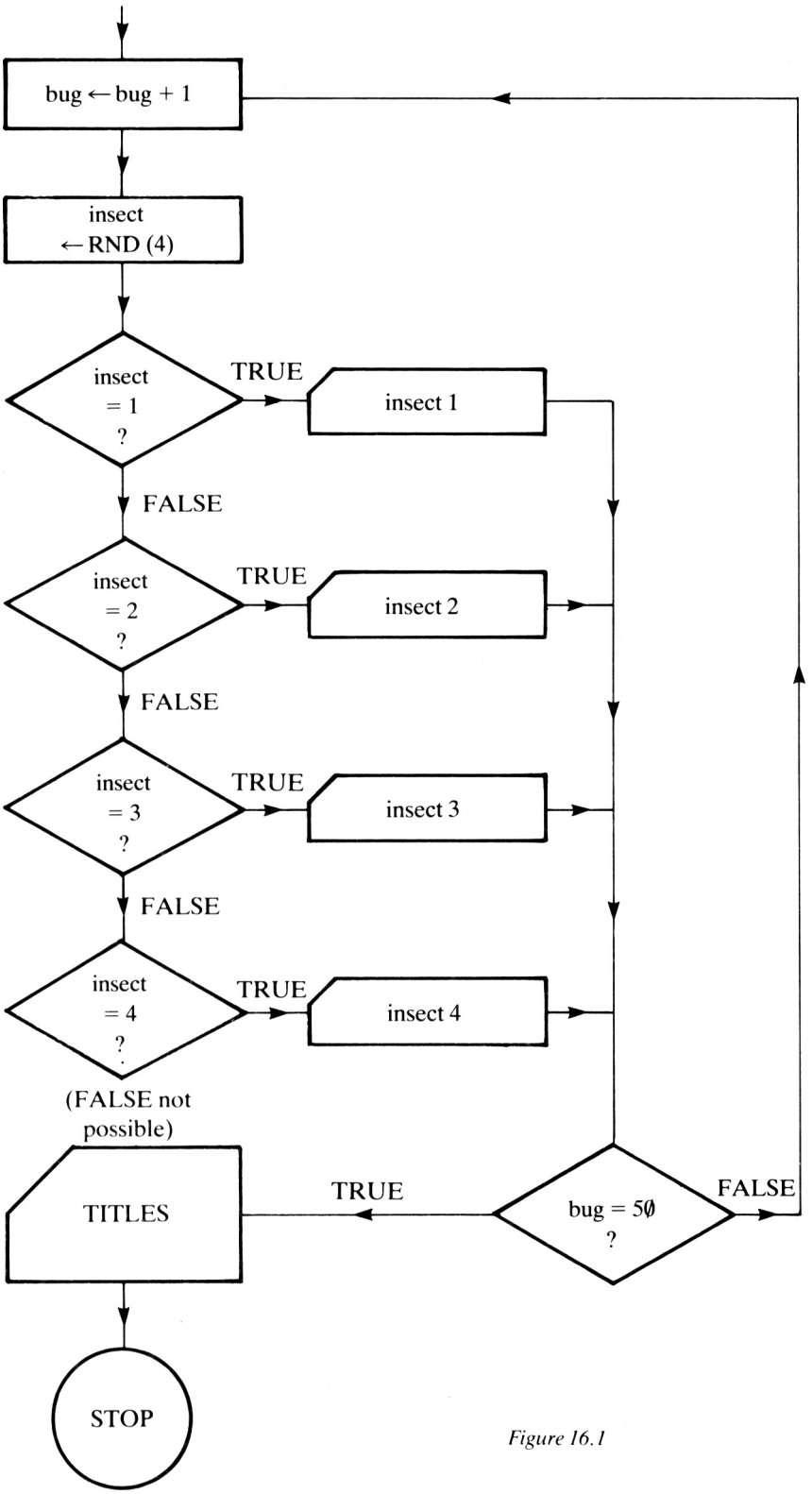

bug ← bug + 1
insect ← RND (4)
insect = 1 ?
TRUE
insect 1
FALSE
insect = 2 ?
TRUE
insect 2
FALSE
insect = 3 ?
TRUE
insect 3
FALSE
insect = 4 ?
TRUE
insect 4
(FALSE not possible)
TITLES
TRUE
bug = 5Ø ?
FALSE
STOP

Figure 16.1

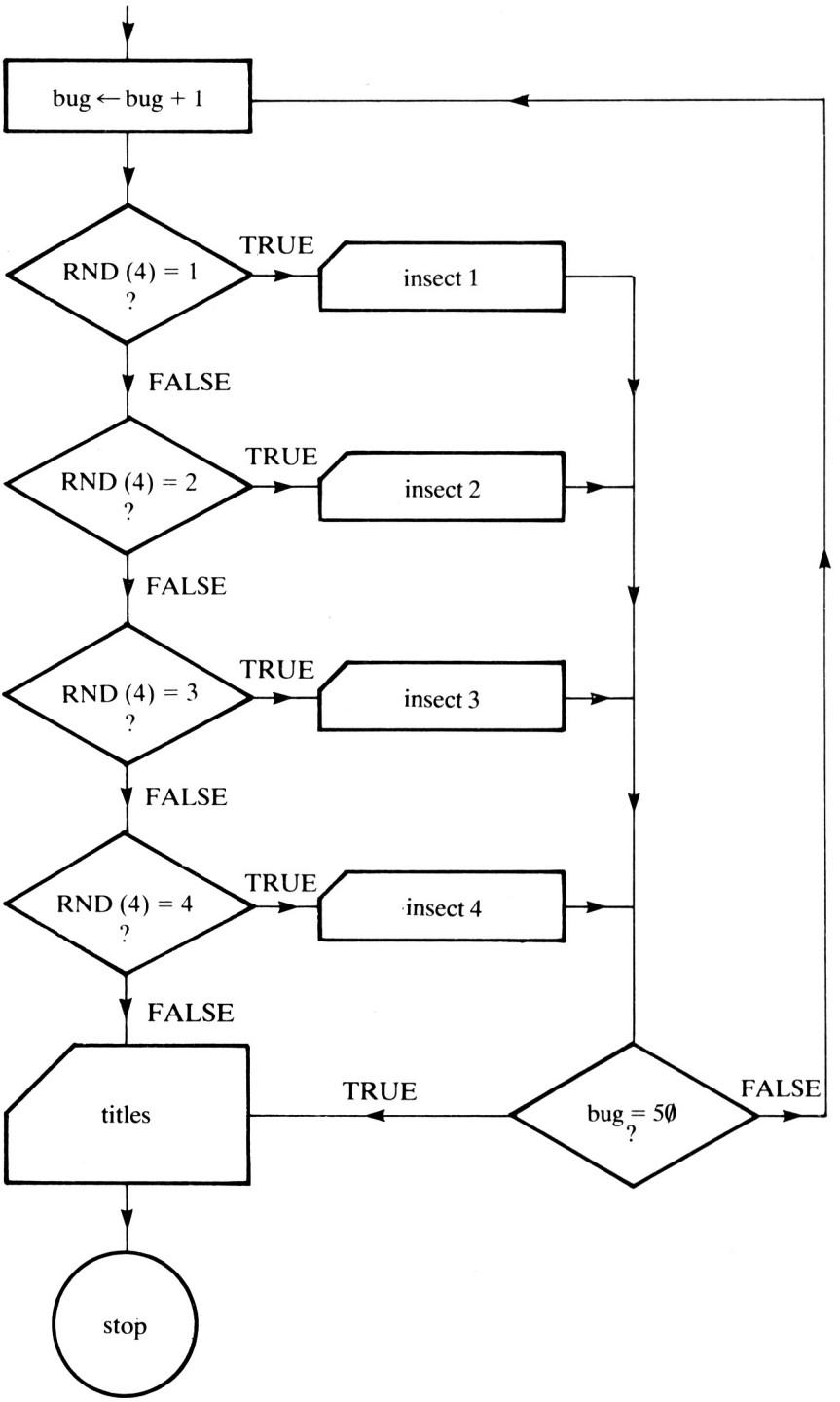

bug ← bug + 1
RND (4) = 1
?
TRUE
insect 1
FALSE
RND (4) = 2
?
TRUE
insect 2
FALSE
RND (4) = 3
?
TRUE
insect 3
FALSE
RND (4) = 4
?
TRUE
insect 4
FALSE
titles
TRUE
bug = 50
?
FALSE
stop

Figure 16.2

Electron BASIC's trace is very useful, when handled with care. It is however sometimes better to use your own TRACE statements in a few selected lines.

Say your program should follow through lines 1ØØ, 11Ø, 17Ø, 18Ø, 12Ø. Put PRINT 1ØØ at the start of line 1ØØ, PRINT 11Ø at the start of 11Ø and so on—then you can see *exactly* what you need to examine. I reckon you won't need the full TRACE facility until you're into really big programs—but practise it and remember it when the time comes.

Anyway all that has given you time to work out what the bug is in *our* program . . . And its cause is tricky—but this is big fault number 3. Can you spot it? Figure 16.1 shows a flowchart of what I aimed at and Figure 16.2 shows a flowchart of what I actually have.

See the subtle problem? The computer isn't choosing a number from 1 to 4 at random and then printing the corresponding insect. Is it? It's choosing a RND(4); if it's 1 it prints insect 1—if not, it chooses again. And so on. So the loop comes to a premature end whenever the micro reaches line 11Ø and chooses a number that isn't 4. Geddit?

How do we get over this problem? There are several ways. The one nearest the flowchart involves adding LET insect = RND(4) at the end of line 6Ø and changing RND(4) to 'insect' in each of lines 8Ø–11Ø. And—hooray—at last we've got somewhere! A nice display of random insects over the screen and the diagonal bugs title.

We've debugged BUGS and got it working as scheduled. We've cleared syntax and typing errors, reminded ourselves of correct loop structures, and seen that a flowchart *does* help program development. We've also emphasized the need for careful screen layout.

There are still problems left, and there are still ways we can structure the program better too. Mainly the routine's too fast—let's slow down the loop then. And it's a shame the title splats some of the insects so fast—a wait loop will solve that. And we could re-start after a while. All that (and more), including a snazzy trick to replace lines 8Ø–11Ø, gives us this polished version. I've kept the original line numbers as much as possible to help you check through with care . . . .

Program 22b: Bunny bugs

```
   10 VDU 23 224 0,84,57,108,254,108,57,84
   20 VDU 23 225 0,5,114,90,202,138,142,0
   30 VDU 23 226 0,82,36,25,62,25,36,82
   40 VDU 23 227 254,130,124,74,90,66,126,0
   45 REPEAT
   50   MODE 2: COLOUR 134: CLS
   60   FOR bug = 1 TO 50: LET insect = RND(4)
   70     PRINT TAB(RND(20)-1, RND(30));
   80     COLOUR insect: VDU 223+insect: SOUND 1,-15,30*inse
ct,2
   90     NEXT bug
  110   LET go_on = INKEY(250)
  120   COLOUR 0: PRINT TAB(0,0);: FOR a = 0 TO 14: PRINT' T
AB(a) " BUGS ": NEXT a
  130   LET g = GET
  140   UNTIL 0
```

SOME CLOSING TIPS

Here are some other debugging tips worth bearing in mind.

1. EDIT in a REM before a statement you don't want used for the time being. If you delete an offending statement, you may need to put it back later—and will you remember it? But beware of REMs in multi-statement lines—the REM will tell the computer to ignore *everything* after it. More perhaps than you bargained for.
2. Statements often worth REMoving in this way are ones with RND. It often helps to have a fixed value rather than a randomly varying one during testing. So think of changing, for instance,

   LET x = RND(1Ø) to LET x = 5: REM LET x = RND(1Ø)

   if you have RND problems.

3. I've suggested before that the last thing you do is to use any ON ERROR statements. However, if you include these before your program is finished and then find that the routines don't work properly, use REPORT. REPORT will then display the last error message as it should have been (but not in as much detail as usual). If there wasn't an error to report, the micro says ESCAPE at you; I'm not sure why.

But the best way to reduce the trials of debugging—and they *are* trials—is not to have bugs in the first place. Well, none of us is perfect—we all make *some* programming mistakes! All the same, the modular, top-down, flowchart-based approach I've discussed earlier is a great help to bug-reduction. And if your program's in modules, you'll find is *much* easier to sort out if it does go wrong. So the moral is—use PROC and bug off . . .

17 Sound as a Bell

The main topic I'd like to introduce in this chapter is the programming of the Electron's sound generation system. That system is complex, and I've no intention of taking it very far—a full analysis would require a book half the length of this one.

As you know, the Electron has its own internal speaker, a small but quite good one at rear left under the cover. It is not difficult to take the signal instead to a more powerful speaker (or two), with or without an amplifier. This does, however, require your working inside the case, and that could invalidate the guarantee. People rarely try to feed sound to the TV speaker, for technical reasons; however it can be done and logically ought to be done.

First let me set out a little background science.

SOUND THEORY

Think of a single note produced by vibration in some musical instrument or other source. We describe it in terms of five factors:

1. *Loudness* (volume)—is it strong or faint? What we call loudness relates to what scientists call *amplitude*. The vibrations are more violent in the case of a loud sound (high amplitude) than a soft one.
2. *Pitch*—is it low or high? Sounds from the left-hand keys of a piano (or a man's throat) are lower pitched than those from the right (or a female voice). Pitch relates to what scientists call the *frequency* of vibration—to obtain a low pitch there are fewer vibrations in a second, in other words, the frequency is low.
3. *Quality* (or *timbre*)—is the sound pure or rough? The difference between the sounds of, say, a trumpet and guitar playing the same note is a difference in quality. What's really happening is that there are different tones, of different amplitudes and frequencies, in each sound.
4. *Attack*—how does the sound start, continue and stop? You may have heard tapes of a piano played backwards—the result is not like a piano at all because the attack is reversed.
5. *Duration*—how long does the sound last?

ELECTRON SOUND

As I said, the sound facility of the Electron is very complex. It allows the user to program all but the first and third of the above factors of a sound. (Most micros, if they have sound at all, allow control only of pitch (frequency) and duration.)

Partly the complexity follows the Electron's offering three channels of sound, which allow separately programmed notes to be 'played' in a certain order. There is in fact a

fourth sound channel—for special effects—which I'll deal with separately. A major source of problems is that the BBC micro (with which the Electron's software has to be compatible) is more complex still.

To control all these factors from instant to instant gives the Electron the potential of a fairly good synthesizer. However it is *not* a synthesizer, so you have to do a lot of work to get anything like real music. Don't be ambitious, at first anyway—stick to simple beeps and warbles, simple sequences of notes and sounds. Don't accept the challenge (given with tongue in cheek in the manual of another micro) to program the whole movement of a symphony—it would be quicker to learn to play the piano!

The Electron has two keywords in this context—ENVELOPE, which broadly controls attack, and SOUND, which causes the speaker to operate. I'll start with the latter. Work through these examples and notes.

1. The SOUND command/statement has this structure (or syntax). (Spaces are optional throughout, but the commas are essential.)

   SOUND C, A, P, T

   (I've chosen the symbols to maximize the memory value!)
   Here C is Channel: Ø for special effects and 1–3 for tones.
   A is Amplitude (volume:) −15 (loud), or Ø (zero volume) or 1–16 for envelopes (see later).
   P is Pitch (frequency): Ø (low) to 255 (high).
   T is Time (duration): 1 (short) to 254 (12½ seconds) or 255 (eternal).
   Explore these effects by giving direct SOUND commands. For the time being keep C = 1 and A = −1. Therefore why not use *KEY1 SOUND 1, −1, (R)?—then you can start off each line with FUNC + 1. Some examples follow:

SOUND 1, −1, 1, 1	[very short buzz]
SOUND 1, −1, 1, 1Ø	[half-second low tone—buzz]
SOUND 1, −1, 1ØØ, 1Ø	[a higher one]
SOUND 1, −1, 25Ø, 1ØØ	[a long squeak]

2. Channel (C) control involves the first of the four SOUND variables. Its value can be Ø, 1, 2 or 3. If you have chosen Ø, you are in the 'noise' channel. But the only way you can get noise out of this machine then, is to use one of the values 4, 5, 6 or 7 as the P value. The details are on Page 118 of the *User Guide*. So SOUND Ø, −1, 6, 1ØØ produces a 5-second burst of 'long period' noise. I'll say a bit more about this later.
3. Amplitude (A) control is really hardly that! There are in fact only two settings of volume on the Electron, sound on (A = −15) and sound off (A = Ø). (The intermediate values will have the same effect as −15, but on the BBC micro actually do produce sounds of differing loudness.) Positive values of A, between 1 and 16, are able to select ENVELOPEs with the same numbers. Again I'll say a bit more about this later.
4. Choice of pitch (P) involves the third of the four SOUND variables. Its value can be between Ø (giving a note somewhat below middle C) and 255 (a little more than five octaves higher). Other values, + or −, are allowed—but they have the same effect as ones in the official range.

   Generally speaking, each step between Ø and 255 corresponds to a quarter of a semi-tone, with middle C (roughly) at 52, and the other Cs at 4, 1ØØ, 148, 196, and 244. (There are 12 semi-tones in an octave.) The SOUND control accepts decimal fraction values of P, but my poor ear can't detect any difference, if there is one, from the corresponding INTegral ones. Thus SOUND 1, −1, 4.95, 1ØØ seems just the same as SOUND 1, −1, 4, 1ØØ.
5. Duration is controlled by T, the fourth variable of SOUND. Ranging from 1 to 254, the value gives the time in twentieths of a second. It seems in practice to be more accurate than the frequency control. (Or is my piano more out of tune than my watch?) T = Ø gives no effect (but on the BBC produces a little click). T = 255, on the

other hand, gives a sound forever—until you let a SOUND statement with a different channel number be carried out, or until you press BREAK or ESCAPE. The first way of escaping from an eternal sound is one spin-off of the three different tone channels (C = 1, 2, 3) that the Electron provides for apparently no purpose. You can use the different channel numbers to schedule the order in which notes are sounded.

If you investigated many of the above ideas, particularly the T = 255 one, you may well have noticed that while a sound goes on you may continue to use the computer. This is a very interesting facility of the Electron, that it can provide 'dual processing'—carrying out SOUND commands while working on other things at the same time.

SYNTHESIS

Because each note must be specified individually, programming tunes can be tedious. No doubt synthesizer software for the Electron will become common but meanwhile, if you want tunes you'll have to put up with the tedium. Well, you could try Program 23 . . .

Program 23: Cynth

(I had to get my wife's name in here somewhere . . .)

```
  10 *KEY1PROCr:SOUND1,-1,52,255!M
  20 *KEY2PROCr:SOUND1,-1,60,255!M
  30 *KEY3PROCr:SOUND1,-1,68,255!M
  40 *KEY4PROCr:SOUND1,-1,72,255!M
  50 *KEY5PROCr:SOUND1,-1,80,255!M
  60 *KEY6PROCr:SOUND1,-1,88,255!M
  70 *KEY7PROCr:SOUND1,-1,96,255!M
  80 *KEY8PROCr:SOUND1,-1,100,255!M
  90 *KEY0PROCrest!M
 101 REM  *   Sorry about the compressed listing, but the E
lk doesn't allow much  room in memory for *KEY work.
 490 END
 491 REM  *   END (or STOP) IS needed this time
 500 DEF PROCr
 501 REM  *   See text about channel priorities
 510 SOUND 2,0,0,1
 520 ENDPROC
 600 DEF PROCrest
 610 PROCr
 620 SOUND 1,0,0,255
 630 ENDPROC
```

Notes

This program sets up the top line of number keys as a simple 'piano' keyboard. Keys 1–8 provide the scale of natural C (no sharps or flats), while key 0 gives a rest.

The best thing for you to do is to explore playing tunes with this little program. But to use it you'll have to keep the FUNC key down; why not try adhesive tape or 'Plastitak'? There's a project suggestion at the end of the chapter that may give you other ideas for this if you want to develop them. Oh, just only little thing—you may like to introduce a synthesizer-like tremolo; then use *FX11, 1 as a direct command. *FX11, 20 will get you back to normal, just about anyway.

While the previous program is very good for letting you mess around at the keyboard, it doesn't enable you to record tunes on tape for the computer to play. *That* is what I meant by a tedious business earlier. Here's the best way I found to go about it:

1. Choose a short, easy tune that you know!
2. Play it out on a piano and write down the order of notes. You'll be able to remember the durations and rests.

3. Set keys as follows for use with FUNC:
   *KEY1 SOUND 1, −15,
   *KEY2 SOUND 1, Ø, Ø,
   *KEY1Ø OLD|M|N LIST|M
4. If you dare, use AUTO.
5. Then program . . . For each note press FUNC + 1 and enter the note-value P (see Table on Page 118 of the *User Guide*), and the duration (say 1Ø per crotchet). Then press (R). For each rest press FUNC + 2, enter the duration and press RETURN.
6. ESCAPE to escape from AUTO if that's what you were using, BREAK to check, RUN to test—and then get polishing.

I did all that, and also used another trick to save time, pre-setting crotchet and quaver durations first, to get this program.

Program 24: Call the RSPCA

```
   10 LET c = 10: LET q = 5
   20 FOR a = 1 TO 2: PROCone: NEXT a
   30 FOR a = 1 TO 3: PROCtwo: NEXT a .
   40 PROC3: PROCone
   41 REM  *   Well, the tune's not perfect, but that's a ni
ce short program!
   50 END
   60 DEF PROCone
   70 SOUND 1,-15,21,c
   80 SOUND 1,-15,13,c
   90 SOUND 1,-15,5,c
  100 SOUND 1,Ø,Ø,c
  101 REM  *   Rest
  110 ENDPROC
  120 DEF PROCtwo
  130 SOUND 1,-15,33,c
  140 SOUND 1,-15,53,c
  150 SOUND 1,-15,49,q
  160 SOUND 1,-15,41,q
  170 SOUND 1,-15,49,q
  180 SOUND 1,-15,53,c+q
  190 SOUND 1,-15,33,c
  200 SOUND 1,Ø,Ø,q
  210 ENDPROC
  220 DEF PROC3
  221 REM  *   Just to show what I said way back - PROC name
s can be numbers. Convenient sometimes in fact.
  222 REM  *   Hardly worth a PROC in this case tho'.
  230 SOUND 1,-15,25,q
  240 ENDPROC
```

Did you notice something interesting when running that (apart from the virtuosity of the music)? If not, RUN again and watch the line below RUN on the screen. As soon as the program got to the last phrase for the last time, the READY prompt appeared.

A major feature of Electron sound is that SOUND statements can queue up in a special section of memory, to be processed in turn while the micro gets on with other things. And *that* gives us the possibility of playing tunes while visual action is miming on the screen. In the case of the BBC micro, it also gives the possibility of chords with multi-channel sound, but alas you'll have to use a Beeb to get those thrills.

ENVELOPE

You should by now have a fairly useful repertoire of SOUND effects. Of course there are plenty more. For example have you tried using RND in your SOUND statements, or putting loudness and duration under variable control?

Like this . . .

Program 25: Psyche

```
 20 VDU 23 224 8,8,8,8,104,88,72,58
 30 VDU 23 225 8,12,10,9,104,120,120,58
 40 VDU 23 226 8,8,8,8,104,120,120,58
 50 VDU 23 227 8,12,10,13,106,121,120,58
 60 VDU 23 228 8,8,8,8,104,88,72,56
 70 VDU 23 229 8,12,10,9,104,120,120,56
 80 VDU 23 230 8,8,8,8,104,120,120,56
 90 VDU 23 231 8,12,10,13,106,121,120,56
100 MODE 2: REPEAT
110   REM  *   MODE 2 for most colours
120   SOUND1,-15,RND(255)-4,RND(5)
130   VDU 19 128 RND(16)-1 0;0
140   VDU 19 7 RND(16)-1 0;0
150   PRINT TAB(RND(20)-1,RND(32)-1) CHR$(223+RND(8))
151   REM  *   Ifyou get all those RNDs, I guess you're OK
with RND.
160   UNTIL 0
```

Actually that started off as a one-liner, and then I was going to give you another example. I daren't develop the second example so that *does* remain a one-liner:

```
FOR a = 1 TO 255: SOUND 1, -1, a, 0.1: NEXT a
```

Using a small variation of that I found that the speaker in my Electron resonates at values of 'a' (P) around 110. That means that sounds with pitches around there will sound louder than others.
However ENVELOPE can form varying sounds much better than that:

```
10  ENVELOPE 4, 2, -1, 10, 5, 50, 50, 50, 0, 0, 0, 0, 0, 0
20  SOUND 1, 4, 0, 10
30  SOUND 2, 4, 100, 20
40  SOUND 3, 4, 200, 30
```

Get ENVELOPE with ENV. (and SOUND with SO.). Interestingly (significantly?) there is no FUNC way to get either—yet you can FUNC RAD and DEG, both as easy to type in full and far less often used.

The above lines show something of the power of ENVELOPE, even if the Electron version is only half as powerful as that for the BBC, as you can guess from that row of useless zeros at the end of the ENVELOPE statement. What the Electron ENVELOPE does is to control sounds by shaping them to a pattern of frequency over time. That is at least partly related to the *attack* I mentioned before. You can believe that, with eight control variables, ENVELOPE is a complex beast to handle. I can't say much more about it here, but you will find plenty of it in magazine articles and more advanced books. Note what the *User Guide* says on Page 121 about those last six variable values.

You can have up to 16 ENVELOPEs in any given program. SOUND statements refer to them by having 1–16 as the value of A. (But of course you can use the −15 or 0 values as before.) My example has ENVELOPE 4, and that number is also the first, A, of the 14 ENVELOPE variables (parameters). In brief, the others (B–N) concern

B	speed at which one goes through the envelope's four parts
C–E	rate of frequency change in parts 1, 2 and 3
F–H	speed at which one passes through each part
I–L	rate of volume change in parts 1, 2, 3 and 4 (BBC only)
M–N	volume levels at ends of parts 1 and 2 (BBC only)

Some of these variables are shown in Figure 17.1.

Practice is the only method of getting on top of ENVELOPE, and as I say you'll need to search through magazine articles and such like for help. The *User Guide* is somewhat

more helpful than normal in this regard, but the matter of ENVELOPE is really an advanced feature of the Electron.

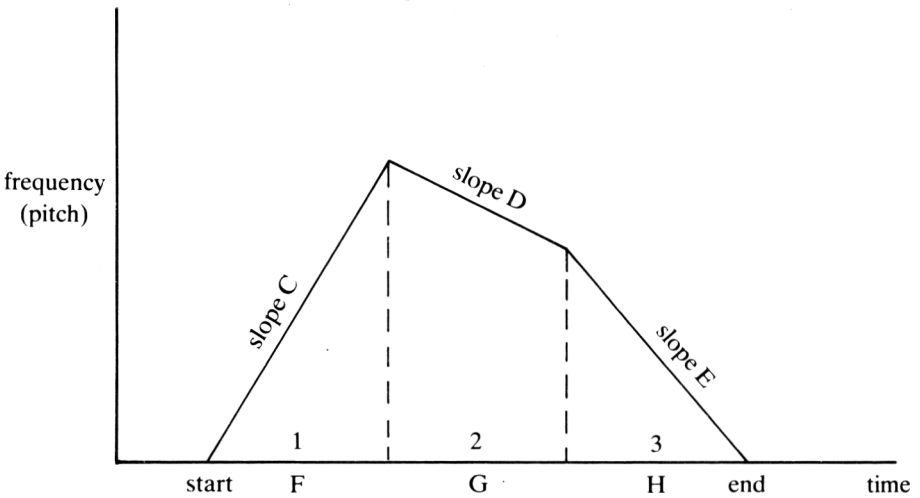


Figure 17.1

CHANNEL Ø

You may have thought we've had special effects already with the ENVELOPE concept. However ENVELOPE + SOUND still gives us only pure notes (fairly pure anyway). What Acorn call special effects sound is produced with Channel Ø (that is the value of the first variable of the four in a SOUND statement). It produces '*noise*'. Noise is a mixture of pure notes like the sound of a radio or TV when there's no station on the setting, or the sound you make called "shh". Here's how we use channel Ø.

```
SOUND Ø, A, F, T
```

The value of 'effect' (F) can be between Ø and 7, but only the values 4–7 are of real interest. Here's a full table (rather like the one on Page 118 of the *User Guide*):

0	high tone
1	middle tone
2	low tone
3	middle tone
4–7	slightly different "shh" noises

Experiments with SOUND Ø, −15, F, 5. Add such lines, amongst other things, to the little ENVELOPE program and see how ENVELOPE statements affect the noises. Sound with the Electron *is* complex, as I started off by saying. It's a fascinating topic, but not one which you can pick up very quickly. I'd certainly advise you not to start to program useable tunes too much—rather design suitable sounds to go with different parts of your program. Those sounds can be fanfares and trills and sound effects, and even snatches of tunes, but don't overdo the sound—you'll lose all your friends!

PROJECTS

1. Try your own 'Cynth' using INKEY$ rather than *KEY. Then you'll be able to try messing with screen colours as I did in Program 25, for instance. Can you match screen colour to speaker output?

2. While ENVELOPE does not control volume (loudness) in the case of the Electron, it is still very powerful. Attempt to develop sounds of vehicles and such like, including, if it turns you on, exploding bombs and 'phasors'. Here's a 'spark', for instance:

    SOUND 0, −15, 6, 20.

And the next goes with the beautiful title 'heavy machinery':

    ENVELOPE 1, 5, 1, 20, 8, 200, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0: SOUND 0, 1, 1, 255.

18 Mishmash

This is the last Utilities chapter in the book, the last one dealing with various programming facilities rather than strictly programming language structures. But I'm not being strict, and shall here look at some aspects of Electron BASIC not dealt with elsewhere.

In particular I shall close the chapter by noting three important areas for the advanced programmer that an introductory book like this cannot attempt to touch. These are interfacing the computer with other equipment; dealing with what are called files; and programming in so-called machine code.

TRUE UTILITIES

First let's look at two commands of use when working on lengthy programs.

DELETE This is a command (not the key at bottom right of the keyboard) we use to remove whole chunks of program from memory. To remove the odd line is easy—you just type the line number and (R). However, you may have the need to remove a set of dozens of lines. This may happen if you'd put the set starting at 3000, then transferred it to 4000 (using EDIT) for some reason. To remove the duplicate set from 3000 would be tedious if you had to use line number + (R). So we have DELETE. Or you may have forgotten to NEW and after entering a few dozen lines of your latest program, want to remove the dregs of the old one. That again would be tedious without DELETE.

The structure of this command is DELETE n, m. Here n is the first line number of the surplus block and m is the last. (Those line numbers don't actually have to exist.) When you press (R) and then LIST, you'll find that all lines between n and m inclusive have gone. Gone for ever, too—you can't get them back with OLD . . .

The structure is thus a bit like LIST n, m, then. However, you cannot use DELETE n, or DELETE, m to remove the first or last parts of a program. This is a pity. To remove everything up to, say, 999, use DELETE 0, 999 (or DELETE n, 999, n being the first line in a program). BEWARE, however, an occasional Acorn bug creeping in—when you delete large blocks from the beginning of a program, the first remaining line may have its number changed to 252 (or, occasionally, some other value like that). To remove everything after, say, 999, use DELETE 1000, 32767 (the highest line number allowed), or DELETE 1000, m (m being the last line in program).
There is no FUNC way to get DELETE, but the abbreviation DEL. will do.

RENUMBER (REN. or FUNC + B) is quite useful but not very flexible. There are four forms:

RENUMBER (R) will give a program starting 10, 20, 30, etc.
RENUMBER n (R) will give n, n + 10, n + 20, etc.
RENUMBER n, step (R) gives n, n + step, n + 2 x step, etc.
RENUMBER, step (R) gives 10, 10 + step, 10 + 2 x step, etc.

Using RENUMBER will give you a new listing with the initial line numbers changed as you have told the micro to do; the addresses of all GOTO and GOSUB statements will be changed to suit the new numbers (unless the addresses are invalid). This includes line numbers in ON statements but does not apply to those mentioned in REMs.

As you know, I like to set out long programs with blocks in different positions in memory (100, 500, 1000, and so on). We can't, however, renumber *part* of a program, so we still need planning. Do not, therefore, get used to REN. as a way of rescuing yourself from a mess! Of course we all have to do that sometimes, but it isn't a good habit.

That reminds me, see what happens if you try to RENUMBER with a value of 'step' greater than 255. I think Acorn's comment here is rather supercilious, don't you?

REM I suppose REM ought really to come before REN., but here it is now.

We've used REM a lot in programs recently, and I trust that by now you are fully into the habit of putting it often into your own programs. The function, you recall, is to insert into a program listing a REMark for the later use of yourself (or others) reading the listing. The computer ignores REMs, but keeps them within the listing (taking up memory) purely for the purpose of humans. Here are some common usages:

```
REM * * fails if a$ = " "
REM * * converts input string to a number if valid
REM * * PROG starts here
```

The use of that spacing and the pretty little stars is partly cosmetic—it helps those REMs to stand out nicely in the listing. (With some computers you can make the REMs flash or appear in a different colour from the remainder, but not in the case of the Electron.)

However, the cosmetic use of REMs is quite important in itself. It is common to separate blocks of programs with several line numbers followed by REM alone: then the blocks stand out boldly. However, there are nicer ways, and I have occasionally used one of them in previous programs in this book. Line number followed by one or more spaces is ignored by the computer, as is line number followed by one or more stars.

The next program shows some of these techniques—but if you run the program you get absolutely no effect!

Program 26: REMbrandt

```
  0 REM  **  PROGRAM 26  **


  1 REM  *   REMs also give instructions to printers (huma
n sort). All the programs in this book carry REMs like the a
bove in the original versions!
 10 REM PRINT'' "What is the point?"


 20 REM  **  End of main program


 30 REM  *   Above I added spaces to the ends of program l
ines. This is an easy way to give fully empty separators, bu
t it's wasteful of memory and time, and not easy to remember
 to do. However I am also using this REM to show how long RE
 40 REMs can be....
 50
 60
 70
 80 REM  *   Line number + " " above
 90 *
100 *
110 *
120 **************************************************
```

```
  130 *
  140 *
  150 * Stars are pretty but take more time than spaces; and
 if you RUN this program now you'd find that SYNTAX would tr
ip you up on this line.
```

Line 10 of the above program is there to remind you of yet another use of REM—to stop the micro carrying out a line that you're not sure about before you actually delete it for ever.

MUG-TRAPPING

That charming name refers to the process of protecting inputs from invalid entries. Other names are input checking, mug-proofing, and entry validation. I'll stick to 'mug-trapping', even though this is a bit intelligencist. What the name means is to trap out all possible errors at input routines, so that even a 'mug' will not be able to make mistakes that cause false data processing or program crashes.

This is all part of the process that I have called *user-friendliness*. You'll recall that by this term I mean making the program crystal clear even to a totally inexperienced user—he or she should never be uncertain what to do, and at all stages should feel comfortable rather than threatened or lacking in confidence.

Part of the user-friendliness thing is to provide clear instruction messages on screen whenever the user is to do anything. Those messages should be clear in language as well as layout, and simple so that any intended user would be able to understand them.

This means that when the program reaches an INPUT routine, there should be more than the "?" prompt on the screen. Some such message should appear as "Please give me a number between 1 and 10."

If you look at that message with care, however, I expect you'll see that it is in fact not crystal clear. Should the user be allowed to enter 1 or 10, or does the programmer really mean 'between'?

So the structuring of messages is an important part of mug-trapping. That is not the end of the process, however. It is entirely possible that an inexperienced user (or even one fully used to computing) may press the wrong key in error, or press RETURN by itself. Other dangers that may appear at a stage like this include the user's entering a number spelt out in letters rather than in digit form. And, finally, there are plenty of people who delight in giving unexpected responses in order to test out a system. (As a teacher I say that with a touch of bitterness!)

So the two stages of data input so far met are not enough; we cannot just have PRINT "message": INPUT input$. The data that is input has to be checked for full validity. It is best to use a PROCedure in order to test inputs suitably. The following program fragments show an example of this.

```
  100 PRINT''' "Now please type a whole number from 1   to 9
"''' "Press RETURN after your answer."'''
  110 REPEAT
  120   INPUT ans$
  130   PROCnumtest(ans$,1,9): IF flag = 1 THEN PRINT'' "A w
hole number from 1 to 9, please!"''
  140   UNTIL flag=0
  990 STOP
 1000 DEF PROCnumtest(t$,min,max)
 1010 LET flag = 0
 1020 LET t = VAL t$
 1030 IF t<min OR t>max OR t<>INT t THEN LET flag = 1
 1040 ENDPROC
```

That example is a fairly simple one. However, the attitude is most important, and I hope that you will see the need as well as the technique for mug-trapping in more complex INPUT cases. Some further ideas appear in the Projects section at the end of this Chapter.

DATA

The facility to 'read' data stored in a program exists in most BASICs. A hangover from the early days of non-interactive computing, it is, in my view, not often very useful now. However, there are occasions when it *is* of real value. Those occasions concern the situations where a program has to process different chunks of data in a certain sequence, and those chunks do not change.

We could have a letter game program, for instance, in which the player has to guess, one by one, a set of letters. Some approaches using existing techniques follow, all very crude demonstrations; it would be a good exercise if you were to try to code each one for yourself from the opening description. First, however, I give PROCguess, the meat of each routine.

```
1000 DEF PROCguess
1010 REPEAT
1020   INPUT'' "What letter do you guess", try$
1030   IF try$ = guess$ THEN PRINT'' "Right!!" ELSE PRINT''
"Wrong!"
1040   UNTIL try$ = guess$
1050 PRINT''''' "Please press RETURN for a new go."
1060 LET g = GET
1070 CLS
1080 ENDPROC
```

1. Using INPUT—a second player enters each letter in secret first.

```
  10 REPEAT
  20   INPUT'' "Player 2: what is the letter to guess", gue
ss$
  30   IF LEN guess$<>1 THEN GOTO 20
  31   REM  *   Simple mugtrap
  40   CLS: VDU 7
  41   REM  *   VDU 7 beeps to tell player to turn round
  50   PROCguess
  60   UNTIL 0
```

   This approach gives maximum flexibility, but is a rotten bore for Player 2.

2. Using a string array—stray? We met arrays briefly in Chapter 15, and I'm going to deal with them fully later.

```
110 DIM store$(20)
111 REM  *   Set up memory space for 20 strings
119 REM  *   Assign strings one by one
120 LET store$(1) = "S"
130 LET store$(2) = "W"
140 LET store$(3) = "K"
150 REM etc
300 REPEAT
310   LET go = 1
320   REPEAT
330     LET guess$ = store$(go)
340     PROCguess
350     LET go = go + 1
360     UNTIL go = 20
361   REM  *   When full list (20) used up
370   UNTIL 0
```

3. Using string processing (Chapter 15)—the method here is to have a single store$, containing all the characters to be guessed and to use string processing to pull out successive characters in turn.

```
410 LET store$ = "OBAFGKMRNSetc"
420 REPEAT
430   FOR go = 1 TO 26: REM if 26 characters
440     LET guess$ = MID$(store$,go,1)
450     PROCguess
460     NEXT go
470   UNTIL 0
```

That in effect uses a version of (2), in which we can view store$ as a one-dimensional character array. Clearly it is much easier to program and much lighter on memory.

4. Using RND to pick out at random one of the characters from the ASCII set.

```
510 REPEAT
520   LET guess$ = CHR$ (RND(26)+64)
530   PROCguess
540   UNTIL 0
```

Here, in an approach even easier to program and lighter on memory, the computer takes over the task of setting the letters to guess. However, it does so at random—and the program in use becomes more tedious more quickly than with any other of the approaches. In particular, there is no method shown above to prevent a 'guess' letter being repeated.

The READ . . . DATA system is almost as easy to code as the third one above, though it is somewhat heavier on memory. It is certainly better than any of the above systems when we come to deal with lengthy strings and/or numbers. Here's our routine now:

```
610 ON ERROR GOTO 650
611 REM  *   See text
620 READ guess$
630 PROCguess
640 GOTO 620
641 REM  *   Tut-tut: 2 GOTOs; more tortuous without
650 PRINT'''' "GAME OVER!!!"''''
660 LET g = GET
670 RUN
680 DATA O,B,A,F,G,K,M,R,N,S,etc
```

This uses two new keywords. READ guess$ (line 620) tells the computer to fetch the first unused item from the DATA list (line 680) and call it "guess$".

We can imagine there's a sort of pointer attached to the DATA list. Each time READ fetches an item from the list, the pointer moves on one place. In our case, if the list carries 20 items (separated by commas, note), the program can use READ 20 times, with different values of guess$ each time.

What happens then? The DATA list pointer has come to the end, so the next READ will have nothing to fetch. That will lead to the error report "Out of DATA" (42)—I've checked this with ON ERROR GOTO 650 in the first line. (See Chapter 9.)

If we want to re-start without RUN, we have to use a third keyword to bring the pointer to the front of the list again—this is RESTORE. It tells the computer to "restore" the pointer to its place at the start of the list. To stick with our letter game I could do this, therefore:

```
610  ON ERROR RESTORE
```

Then the game will go on for ever (until BREAK), with the same set of letters coming up again and again in the same sequence.

However, we can add a degree of flexibility by using RESTORE n, where n is a line number. Each time RESTORE n is met, the pointer will move to n and the next READ will dig up the first DATA item listed thereafter.

Now, then, we can have the 26 possible letters in 26 separate DATA statements and access them at random with a RESTORE (random n to n + 26) statement in rather the same way as method (4) above. That gives us this version of the game, which I have improved significantly to make it easier to play.

Program 27: Letter spray

```
10 MODE 1: VDU 19 128 3 0;0 19 3 4 0;0
11 REM  *   See how we can combine VDUs
20 REPEAT
30   RESTORE (99+RND(26))
31   REM  *   RESTORE is FUNC+W or RES.
```

```
   32   REM  *   RENUMBER won't be happy here
   40   READ guess$
   50   PROCguess
   60   UNTIL Ø
  100 DATA A
  101 DATA B
  102 DATA C
  103 DATA D
  104 DATA E
  105 DATA F
  106 DATA G
  107 etc
  170 etc
 1000 DEF PROCguess
 1010 REPEAT
 1020   INPUT'' "What letter do you guess", try$
 1030   IF try$ = guess$ THEN PRINT'' "Right!!" ELSE IF try$
< guess$ THEN PRINT'' "Too low..." ELSE PRINT'' "Too high..
."
 1040   UNTIL try$ = guess$
 1050 PRINT''' "Please press RETURN for a new go."
 1060 LET g = GET
 1070 CLS
 1080 ENDPROC
```

Most uses of READ . . . DATA you may meet have little, if any, advantage over either INPUT or (LET). The above one doesn't, for instance—until the need is to guess names or phone numbers, for instance, anyway. (Most 'hangman' games of medium sophistication use DATA.) However using the system with the RESTORE n facility makes it powerful in a number of 'serious' games situations.

A couple of more little points about READ . . . DATA. First note lines 100 onwards in Program 27: there is no need to distinguish between strings and numeric data—we don't need quote marks in the former case. This unique Electron feature is very helpful, even if theoretically dangerous. (You *can* put string data in quotes if you like—the machine doesn't mind—and you *must* do so if the string in question starts with one or more spaces or contains a comma.)

READ is fairly flexible. You can follow it with one or more variable labels (string or numeric—but they must match the DATA items accessed), separated by commas as with INPUT:

READ name$, address$, phone

The main use of the Electron in which READ . . . DATA is near essential concerns the playing of tunes. A long tune will use a huge amount of memory if set as in our previous chapter, as it consists of separate lines like SOUND 1, −15, etc. Even if you enter these in the short way described on Page 110, it'll take time and gobble up memory. Try this then. It gives an example of the READ a, b structure I just mentioned.

Program 28: Briny

```
   10 ON ERROR PROCwait: GOTO50
   11 REM  *   The expected error is Out of DATA
   20 VDU 23 223 64,160,17,17,17,17,10,4
   30 MODE 5: VDU 19 128 2 0;0 19 3 4 0;0: CLS
   40 PROCnotes
   50 RESTORE: REPEAT
   60   READ pitch, time
   70   IF pitch<>0 THEN SOUND 1,-15,pitch,time*4 ELSE SOUND
 1,0,0,time*4
   71   REM  *   Pitch 0 is for a rest
   80   FOR a = 1 TO 6: PRINT TAB(RND(20)-1,RND(32)-1) CHR$
223: NEXT a
   90   UNTIL 0
   98 REM  *   Here starts the music score for the micro to
READ
   99 REM  *   Spaces not needed
  100 DATA E,3, G,1, A,2, G,1, E,1, D,1, C,1, B,1, C,1, G,2,
 G,2, 0,2
```

```
  110 DATA FS,1, G,1, A,2, G,1, E,1, D,1, C,1, B,1, C,1, A,4
, 0,2
  120 DATA GS,1, A,1, C`,2, B`,1, C`,1, G,1, E,2, 0,2
  130 DATA C`,1, B`,1, G,2, FS,1, G,1, E,1, C,2, 0,2
  140 DATA FS,1, G,1, B`,2, A,1, G,1, F,2, E,1, D,1, C,4
  200 DEF PROCnotes
  210 LET B = 48
  220 LET C = 52
  230 LET D = 60
  240 LET E = 68
  245 LET F = 72
  250 LET FS = 76
  251 REM  *   That's F sharp
  260 LET G = 80
  270 LET GS = 84
  280 LET A = 88
  290 LET B` = 96
  291 REM  *   That's B in upper octave
  300 LET C` = 100
  310 ENDPROC
  500 DEF PROCwait
  510 FOR wait = 1 TO 4000: NEXT wait
  520 ENDPROC
```

It helps to have the *KEY settings from Program 23 available when writing tune material like this!

The traditional uses of READ . . . DATA are associated with record-keeping programs, such as that represented by the following fragment, part of a record collection file system.

```
 1000 DEF PROCsearch
 1010 INPUT' "What group", group$
 1020 PROCspace
 1030 FOR a = 1 TO 50: REM  *  If 50 records
 1040   READ record$
 1050   IF LEFT$(record$,20)=group$ THEN PRINT' group$' TAB(
3) MID$(record$,21,20)' TAB(5) RIGHT$(record$,20)
 1060   NEXT a
 1070 ENDPROC
```

This implies that the details ('record') of each record are held as a DATA line in the form of a 60-character string. This has three parts (group, title, things like label and date). Each part *must* contain 20 characters, so you would need to fill up with blanks if required (PROC space would do that). Each data statement would have the form

DATA GROUP (15 spaces) TITLE (15 spaces) NOTES (15 spaces)

as in DATA PINK□FLOYD□□□□□□□□□□□OVER□THE□ELK

□□□□□□□□MICROTUNES□985□EP13

This is fairly unwieldy, though it would work. It would be better to use arrays (Chapter 20) in practice. Or even FILES.

FILE-HANDLING

I have no intention of saying very much about this rather advanced aspect of Electron BASIC here. It is not widely used outside certain specialist software areas.

A *file* is any collection of data held in cassette (or otherwise) and able to be called upon by the process unit of a computer. So BASIC programs saved on cassette are really files—we access them using LOAD or CHAIN. In this context, though, we tend to mean by 'file' a set of data held in store for use by a program rather than the program itself.

Electron file-handling is designed for the BBC disc system. Most users will stick with cassettes, at least for the time being—they are far cheaper, although much slower—so

please realize that file-handling in this case is not fantastic. In particular, note that anything but the most simple uses certainly require your cassette machine to be under the control of the Electron.

Anyway, here are the outline details of what is not really a very complex process. Compare the language I use throughout with how you handle paper files with a filing cabinet system.

1. The first thing to do is to create a file, opening a new one in other words. The structure involved (as command or program line) is LET f = OPENOUT "name".
   Here f can be any numeric variable and "name" any string. For instance

```
1000  INPUT "File-name", f$
1010  LET f = OPENOUT f$
```

   The micro is now ready to send data out to the cassette/disc as a file. ("OPENOUT" = "open for output".)
2. Writing to a file—this means sending data. The structure is very like that of a mixed PRINT statement (except you can use only comma separators—not to be replaced with spaces):

```
1020  PRINT # f, "example"
```

   The hash symbol # (on the 3 key) tells the computer which file to write to (but in cassette filing you can write to only one at a time). # is a common US symbol for "number".
3. Closing a file—this must be done when no more data is to be sent.

```
1030  CLOSE # f
```

4. Reading from a file—meaning recapturing recorded data. First the file must be re-opened—this time, as the file already exists, by using OPENIN (or OPENUP) rather than OPENOUT. ('OPENIN' means 'open for input'.) Then we use INPUT for the actual reading action:

```
2000  LET f = OPENIN "name"
2001  REM * the same or a different variable may be used this time—but you
      MUST of course call the right name!
2010  INPUT # f, i$
2020  CLOSE # f
```

5. Other related keywords are

BPUT #	write a single byte to the file
BGET #	read a single byte from the file
IF EOF #	. . . check for the End Of the File signal.

Computer data files have potential in various situations where paper files now rule! In particular, a file could replace the card index active hobbyists use with their collections; while if I were organised enough to produce this book by 'word-processing', data (text) files could replace great sheaves of paper. But I ought to repeat that Electron filing is fairly feeble unless you have the use of discs, or at least 'floppy tape'.

MACHINE CODING

Like most micros, the Electron can accept instructions in low-level code rather than high-level BASIC form. It has what is called an *assembler* facility for this purpose. Such work is fairly advanced and I shall say little about it here. However there are major advantages of assembler programming; in particular such routines may work dozens of

times faster than the corresponding BASIC ones.

The Electron has readily available inside its little chips what is called an '*assembler*'. This is a (machine code) program which can translate into machine-code the instructions you give it in a fairly low-level language called 'assembly code'.

Assembly code sections of BASIC programs must open with "[" (using SHIFT + COPY) and they must close with "]" (using SHIFT + CTRL + COPY).
Relevant keywords are:

DIM name n
LET P% = name — reserve n bytes of memory for the section called "name"
CALL name — tells the micro to carry out that section, = PROC or GOSUB

The next program is a very simple example. As you see, it is in essence a BASIC program, with imbedded in it (lines 5Ø–1ØØ) some strange new gibberish. That imbedded material is an assembly coded program. Let's get it into the computer and see what happens. Best switch off and switch on again (not normal practice). And then, of course, re-enter your usual *KEY 1Ø line.

Program 29: Assembly bell

```
 10 VDU 23 223 16,56,56,56,56,124,254,32
 20 DIM a 20
 30 LET P% = a
 40 [
 50 .go LDA #32
 60 JSR 65507
 70 JSR 65507
 80 LDA £223
 90 JSR 65507
100 JMP go
110 ]
```

When you have entered this program, check it very carefully indeed—errors in machine code programs can have catastrophic effects . . . .

Satisfied? So—RUN. What happens? Nothing special. You get a new listing of the program. This listing is in machine code, and I reproduce it as Figure 18.1.

```
ØE96
ØE96   A9  2Ø      .go   LDA # 32
ØE98   2Ø  E3  FF  JSR   655Ø7
ØE9B   2Ø  E3  FF  JSR   655Ø7
ØE9E   A9  DF      LDA   # 223
ØEAØ   2Ø  E3  FF  JSR   655Ø7
ØEA3   4C  96  ØE  JMP   go
```

Figure 18.1

So running Program 29 has the effect of carrying out all BASIC instructions (though you don't know that yet), and 'assembling' the assembly code instructions into machine code. The system then prints up on screen the machine code instructions with their 'line numbers' (actually memory addresses) and, to the right, the assembly code material from which they were translated. Whenever you want to carry out this machine code routine use CALL name; in this case the name is 'a'—so use CALL a (R). And, lo and behold your machine code program is running. It will run for ever as the original assembly code routine was a closed loop. Press ESCAPE to escape—but it has no effect. Machine code is a lot less friendly than BASIC! In fact the only way you can escape from this is to press BREAK and when you do this you will get back a listing of your original program if you had programmed that BREAK key in the usual way.

What the above simple routine does is simply to use machine code to print the pattern you saw on the screen. I don't actually think the effect is much faster than what you'd get if you used a simple BASIC loop, but there are plenty of occasions where speed does increase greatly.

Let's go through the listing of Program 29 in a little more detail.

Line 10 A standard BASIC instruction to design CHR$ 223 as a 'bell'.

Line 20 DIM a 20—reserves 20 bytes of memory for a machine code routine with the name 'a'.

Line 30 LET P% = a puts the name 'a' into a special Electron variable (an integer one, hence the %) used for the purpose.

Line 40 The square bracket shows that we are going to start using assembly code.

Line 50 '.go' is the label for this line; I'll refer to it again in a moment. LDA 32 means copy into a special part of memory called the accumulator the character with number 32—the space. 'LDA' is the assembly code instruction for 'load into the accumulator'.

Line 60 JSR 65507—jump to the subroutine stored in memory at location 65507. That particular section of memory, part of the Electron's operating system, is a subroutine for printing the contents of the accumulator onto the screen.

Line 70 As line 60.

Line 80 As line 50—but this time the character loaded is the one coded 223, the one specially defined in line 10.

Line 90 As line 60.

Line 100 JMP go—jump to the line whose label is 'go'.

Line 110 The square bracket shows that the assembly code routine is finished.

I cannot in this book, let alone in only a couple of pages of it, explore in any more detail the intricacies of machine coding. Programming with this system is exciting and valuable —but it is a totally different ballgame to BASIC programming, and I guess we'd better get BASIC programming sorted out first. No doubt, soon after this book has appeared, others will be published dealing with nothing else than assembly coding for the Electron.

INTERFACING

Here we think of using the micro to capture information from outside devices and/or to control outside devices. Examples of the two types are photocells; electronic thermometers and rooftop radar systems; and lamps, motors, and nuclear power stations.

The only interfacing *you* can now carry out with your Electron is to control the cassette recorder if your cassette recorder is one that can be controlled. It is in fact possible to use the cassette control system to some extent for handling other devices. Again, however, that is advanced, and I'll leave you to find out about that in your own good time, by scanning suitable magazines for articles on the subject.

For more sophisticated work than that you will have suitable 'interfaces' connected to your micro—they will no doubt involve using the strange thing sticking out of the back of the machine. At the time of writing, though, there is no mention of such work or of such interfaces, and no-one I know is even sure how to use the Electron to drive a printer.

Uses of interfacing are very exciting, all the same, and I hope you'll be able to explore some later on. They include:

controlling tape-slide equipment from an interactive program;
burglar alarms and decoys;
flashing light displays;
central heating control with an array of sensors.

The Electron as it stands is not really designed for such work, however—and if you fancy that to be your scene, maybe you will need to consider another machine in due course.

19 Graphic Phrases

And now we come to the last major area of Electron BASIC programming—graphics. This includes all aspects of 'drawing' and 'painting' as carried out by a colour computer—producing points, lines and shapes, and blocking in areas with solid colour. As is the case with Electron sound programming, the graphics facility is very sophisticated and therefore not always straightforward, in part to allow compatibility with the BBC micro.

MODES

Of course you know that your computer has a number of different 'modes' of action. When you switch it on, you get MODE 6. And you can call on six others, quaintly numbered Ø–5. Most of the work done in this book so far has been set in MODE 6 (the starting mode) or MODE 5 (the graphics mode closest to it). But here is a full list of the seven modes (like that on Page 31).

Table 19.1

Mode	Text	Zones	Colours	Graphics	Memory/K	Speed
Ø	32 × 8Ø	8	2	256 × 64Ø	2Ø	slow
1	32 × 4Ø	4	4	256 × 64Ø	2Ø	slow
2	32 × 2Ø	2	16	256 × 16Ø	2Ø	slow
3	32 × 8Ø	8	2	—	16	so-so
4	32 × 4Ø	4	2	256 × 32Ø	1Ø	fast
5	32 × 2Ø	2	4	256 × 16Ø	1Ø	fast
6	25 × 4Ø	4	2	—	8	fast

The 'text' column shows the number of lines of text allowed and the number of characters allowed in each one. For text we measure (using 'TAB') from position (Ø, Ø) at top left of the screen. The print zones (Chapter 5) contain ten character positions each, hence the data in the third column. In the 'colours' column I show the number of colours allowed on the screen at any one time (see Chapter 7). Then there are the 'graphics' data, giving the number of vertical and horizontal points you can plot at. The micro will ignore your attempts to plot in MODE 3 or 6 (though here, as in all the graphics modes, you can use your own characters designed with VDU 23—Chapter 5 again).

The next column tells you how much memory the micro reserves for its own strange needs as soon as you enter a given mode. The more complex a display can be, the more memory is reserved in this way. The maximum allowed is 32K, so taking the value in the memory column from this tells you the most memory you have left for your programs in a given mode. (Recall that the unit of memory size is the 'K'. This stands for 1Ø24 of the memory units called *bytes*, each byte being eight bits if you know about such things.)

You call the mode you want, as you know, by MODE. Get that with MO. or FUNC + M. The instruction can be used as a direct command, or as a statement within the program (but *not* in a procedure). MODE such and such clears the screen, by the way. All the same, I suggest you always use CLS after it to remind you to think about layout, and you'll have to do that if you want anything else than white ink on black paper—this is because the MODE statement always gives you a black screen with white bits on unless you tell the micro otherwise.

THE DRAW COMMAND

As you may guess, this tells the micro to draw a line—but as it is a graphics command, it's ignored in MODES 3 and 6. What it does is to draw a line from some previous point to the one whose position (coordinates) you tell it after the keyword, DRAW.

We need a bit of background knowledge to get the hang of the coordinates concept. Look at Figure 19.1. The screen is always (whatever the mode) 'understood' by the computer as consisting of 1Ø24 lines (y-values) each of 128Ø points (x-values). It is a bit imaginary as no Electron mode currently available can actually access such small points, and no TV set could show them. All the same, Acorn are (as well as making things hard) planning for the future.

Anyway, like the simple graphs you've probably drawn at school, we have a co-ordinate system here, based on an origin (x = 0, y = 0) at the bottom left corner of the graphics area. (When you become an advanced programmer, you will learn how to re-site the origin when you want.)

Point A in the Figure is 1ØØ steps across and 3ØØ steps up. Its x, y coordinates are (1ØØ, 3ØØ) and if we use DRAW 1ØØ, 3ØØ we should get a line to it. Where's the line from? If the micro isn't told otherwise, it'll draw the line to A from the origin. Switch on and try it. DRAW is DR. or FUNC + D.

No result? Ah, when you switch on, or use BREAK, you're in MODE 6, which ignores graphics instructions. Go to MODE 5 and try again. OK? A steep staircase ('straight line') from the origin to A.

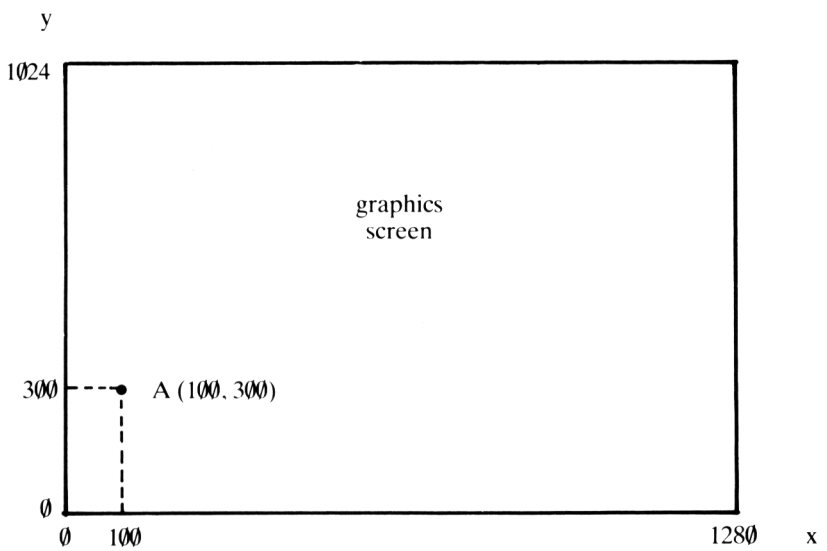


Figure 19.1

The next program gives you the basis for using DRAW in each mode. I'll explain it after the listing.

Program 30: Your move

```
 10 FOR mode = 0 TO 6
 20   MODE mode
 30   PROCdraw
 40   NEXT mode
 50 END
 60 DEF PROCdraw
 70 CLS: VDU 5
 80 DRAW 100,300
 90 PRINT "A"
100 MOVE 100,300
101 REM  **  No FUNC or useful abbreviation here
110 DRAW 700,700
120 PRINT "B";
130 DRAW 10,10
140 PRINT "C"
150 INPUT "x(<1000), y(<1000)", x,y
160 MOVE x,y: PRINT "Mode "; mode
170 LET g = GET
180 ENDPROC
```

RUN the program and note how the fineness ('resolution') of the drawn lines varies, yet the scale doesn't—even if the character size changes.

Notes

VDU 5 (line 70) lets you mix print (text) and lines (graphics) fully. It hides the cursor in the graphics modes, and lets you print literally anywhere (as in lines 140–160).

DRAW 100, 300 (line 80) gives a line from 0, 0 to that point, A. We then (line 90) print 'A' starting at that point.

MOVE 100, 300 (line 100) is to get us back to point A after doing the printing. See the difference at point B as a result of the different structure of line 120: PRINT rules are still followed!

Play around with this program. Note with care the effects of different PRINT positions (as in line 140 above) and the effects of moving or drawing or printing outside the 1280 × 1024 screen. Become aware of the features of the different graphics modes you have available. Try the program without VDU 5.

Explore the next program in the same sort of way. This also extends the technique of colour control, using VDU 19 . . . in the graphics modes.

Program 31: A real cat's cradle?

```
 10 MODE 1: ON ERROR GOTO 120
 11 REM  *   Traps ESCAPE key
 20 VDU 19 128 10 0;0
 30 VDU 19 3 11 0;0
 31 REM  *   Or combine the two with VDU 19 128 10 0;0 19
3 11 0;0
 40 CLS: VDU 5
 50 LET line = 0
 60 REPEAT
 70   LET x1 = RND(1280): LET x2 = RND(1280)
 80   LET y1 = RND(1024): LET y2 = RND(1024)
 90   MOVE x1,y1: DRAW x2,y2
100   PRINT line: LET line = line + 1
110   UNTIL 0
120 VDU 19 128 1 0;0
130 VDU 19 3 0 0;0
131 REM  *   Or combine as Line 31
140 REPEAT: UNTIL 0
```

On the basis of what we've done before, are you happy with lines 9Ø–1ØØ? I hope so.

But it's now time to say more about the VDU 19 instruction, which I've used a lot already in fact. This is the way to get non-standard colours in the various modes. Non-standard? Well, I guess I should first say what the *standard* colours are. . . .

Folk often call MODEs Ø, 3, 4 and 6 two-colour modes—only two colours can be on screen at once. These bear the 'logical' colour numbers 128 and 1 and normally give black background (paper) and white foreground (ink) respectively. See Table 19.2.

Table 19.2 'Logical' colours in MODEs Ø, 3, 4 and 6

Paper	Colour	Ink
128†	black	Ø
129	white	1†

In that table (and the ones that follow), the dagger (†) gives the 'default' values—the ones that the micro uses unless you tell it otherwise. MODE 1 and 5 allow four colours on the screen; their standard 'logical' values are shown in Table 19.3.

Table 19.3 'Logical' colours in MODEs 1 and 5

Paper	Colour	Ink
128†	black	Ø
129	red	1
13Ø	yellow	2
131	white	3†

You first met these values in Chapter 7, when we looked at the COLOUR statement. COLOUR lets you say what ink and paper values you want to use out of those available in the MODE you are in.

Thus in MODE 6 (the switch-on mode), using COLOUR 129: COLOUR Ø will give you black text (ink) on white paper. (You also get for free the strange 'railway line' effect of Mode 6.)

MODE 2 is a sixteen-colour mode (or eight, depending on your point of view). The logical numbers and their values are as in Table 19.4.

Table 19.4 'Logical' colours in MODE 2

Paper	Colour	Ink	Paper	Flashing colour	Ink
128†	black	Ø	136	black/white	8
129	red	1	137	red/cyan	9
130	green	2	138	green/magenta	10
131	yellow	3	139	yellow/blue	11
132	blue	4	140	blue/yellow	12
133	magenta	5	141	magenta/green	13
134	cyan	6	142	cyan/red	14
135	white	7†	143	white/black	15

In any mode, you will start up with the daggered colour values. You can then use COLOUR to change paper and ink to another pair from those available if you so wish. In MODE 2, this means that you can use COLOUR to select any of the available sixteen arrangements.

In any other mode, VDU 19 . . . lets you change an allowed logical colour number to a new one from the full range of sixteen. Its structure is VDU 19, logical colour number, required colour number, 0, 0, 0 or VDU 19 logical colour number, required colour number 0; 0. The Electron's required colour numbers are the same as the ink colour numbers in MODE 2.

Lines 20 and 30 in Program 30 use this VDU 19 thing. They make the foreground and background take new colours not standardly available in MODE 1; both of the new 'colours' are in fact flashing arrangements. Once those new colours have been defined they will be used until you change them or press BREAK. That means that if you ESCAPE from the above program, the line-drawing will cease, but the screen will continue to flash. To avoid that problem I trapped the use of ESCAPE in line 10—when you want to stop the program using that key, the ON ERROR statement sends us to line 120 and onwards, and there the VDU 19 calls there come into action. They stop the flashing and 'freeze' the picture for you to examine in peace and quiet.

Line 140, as usual, prevents the 'finished' cursor from appearing. However, this time the error trap in Line 10 means that only BREAK will get you out! We call this 'disabling' the ESCAPE key; I've already noted how to disable BREAK.

USING PLOT

PLOT is the most versatile graphics command. The MOVE and DRAW we've already met are in fact special names for the two most common of its 60 or so styles. (There are to be more styles in later extensions of the Electron.) The structure is

PLOT s, x, y (commas needed, spaces optional).

Here 's' is the style number, and 'x' and 'y' are as before. The allowed values of 's' provide various facilities. I summarize these in the next few lines, and explain in more detail after.

Move to x, y
—from the previous point (+0), or from the origin (+4);
—without drawing (+0), drawing a continuous line (+1), drawing a dotted line (+16), leaving out the last point in a line (+8);
—using no colour (+0), foreground colour (+1), the background colour (+3);
—marking the point x, y with a dot (+64), or not (+0);
—filling the triangle between x, y and the two previous points (+80) or not (+0).

Any PLOT action you want must be described by choosing just one statement from each of the above five lines. Then you get the action you need by making the value of 's' the sum of the corresponding numbers. Struggle through the heavy seas of Program 32 in order to see what I mean a little more clearly.

Program 32: Morning Cloud

1. Set up, and get to 200, 700, having first gone to 100, 100:

```
10 MODE 1: VDU 5
11 REM  *   Hide cursor
20 VDU 19 128 6 0;0
21 REM  *   Cyan paper
30 VDU 19 3 1 0;0
31 REM  *   Red ink; COLOUR 1 would do
40 CLG
41 REM  *   Truly clear screen
50 MOVE 100,100
60 MOVE 200,700
```

2. To plot a triangle between those two points and 1000, 100:

Measure from origin	+4
Without drawing	+0
In ink colour	+1
Without dotting	+0
Fill triangle	+80
	85 = s

```
70 PLOT 85, 1000,100
```

3. Draw a dotted vertical line down from the second point:

Measure from origin	+4
Draw dotted line complete	+16
Background colour	+3
No extra dot or triangle	+0
	23 = s

```
80 MOVE 200,700
90 PLOT 23, 200,100
```

4. Take that to the edge of the screen:

Measure from last point	+0
Draw dotted line complete	+16
Foreground colour	+1
No extra dot or triangle	+0
	17 = s

```
100 PLOT 17, 0,-100
101 REM  *   100 points down
```

5. Print two cyan triangles inside the main triangle, in each case as follows, to make a square:

Measure from last point	+0
Without drawing	+0
Background colour	+3
No dot	+0
Fill triangle	+80
	83 = s

```
120 MOVE 250,400
130 MOVE 350,400
140 PLOT 83, -100,100
150 PLOT 83, 100,0
151 REM  *   Go thru that useful technique with care!
```

6. Finish off—see if you can sort out the values of 's'!

```
160 PLOT 0, -70,-15
170 PRINT "G"
180 PLOT 4, 280,450
190 PRINT "B"
200 REPEAT: UNTIL 0
210 REM  *   Keep out final cursor
```

I called that 'Morning Cloud' because it's like part of a yacht (and that's the name of one). I hope it's not too foggy. I hope that you can now take the idea further as you explore PLOT, and I suggest some ideas based on that program in Project 3. Meanwhile here are some particularly useful versions of PLOT s:

PLOT Ø,	Move from last point and do nothing
PLOT 4,	= MOVE (from origin)
PLOT 5,	=DRAW (foreground colour line from last point)
PLOT 69,	Put a dot in position in foreground colour
PLOT 71,	Remove a dot (= plot in background colour)

Now you can really try to come to grips with the marvellous 'patterns' program in the Introductory Cassette. (Program A4 at the end is a posher version of that.)

TROOPING THE COLOUR

We have so far met two ways of getting coloured foregrounds and backgrounds in MODEs Ø–6. The first uses the COLOUR keyword, which is designed for use with text material. The second, VDU 19 . . ., applies to graphics operations (including any text on the screen).

VDU 19 . . . is rather cumbersome. It can be abbreviated a little, as you know, by using Ø; Ø at the end but that doesn't make much difference.

There's a third colour command, GCOL (= graphics colour). GCOL Ø, is followed by a number with just the same rules as the colour numbers in COLOUR statements. Thus the second colour statement at the start of Program 32 could have been written like this:

```
3Ø  GGOL Ø, 1 (red ink)
```

and you could add 11Ø GCOL Ø, 2 to change the ink to yellow for the GB plate.

GCOL is simpler than VDU 19 . . ., but allows far less flexibility at our level. I suggest you use COLOUR statements in text-only routines, and choose between VDU 19 . . . and GCOL Ø, . . . as you need! You will certainly prefer GCOL if you want to get more than two colours on screen at once without too much hassle while plotting.

Anyway, when in graphics mode, use CLG (= clear graphics area) rather than CLS(= clear screen). This is particularly important if you use VDU 24 to define a graphics area ('window') within the whole screen. (See *User Guide*, Page 110.) The CLG would clear that area while leaving the remaining (text) area untouched.

Now is the time to mention VDU 29 . . . . This is used to change the graphics area origin coordinates from Ø, Ø, to x, y. In this case refer to the *User Guide*, Page 113. I suspect that the main use of this is when drawing graphs which do not remain within the +x, +y region, but you'll also need it in graphics windows made with VDU 24.

Let's be traditional, and have a line-graph drawing program. Then maybe you can see where change in the origin could be of use in certain circumstances.

Program 33: On the line

```
   10 MODE 1: COLOUR 130: COLOUR Ø: CLS
   20 PROCenter
   30 PROCscale
   40 PROCdraw
   50 LET wait = GET: RUN
   60 DEF PROCenter
   70 PRINT TAB(Ø,5): INPUT "How many points", set: DIM xset
(set): DIM yset(set)
   71 REM  *   Arrays; see next Chapter
   80 CLS: PRINT'' "Type x, y RETURN for each point."''
   90 FOR point = 1 TO set
  100   PRINT "Point "; point "- x,y:"
  110   INPUT xset(point), yset(point)
  120   NEXT point
  130 CLS
  140 VDU 14: FOR point = 1 TO set
```

```
  150   PRINT; point " x = "; xset(point) " y = "; yset(poin
t)
  160   NEXT point
  170 REPEAT
  180   INPUT "Errors? Y(es) or N(o)", check$
  190   IF check$ = "Y" THEN INPUT "Which point", point: PRI
NT "Point "; point "  x,y: ": INPUT xset(point), yset(point)
  200   UNTIL check$ = "N"
  210 ENDPROC
  220 DEF PROCscale
  230 LET xmax = 0: LET ymax = 0
  240 FOR point = 1 TO set
  250   IF xset(point) > xmax THEN LET xmax = xset(point)
  260   IF yset(point) > ymax THEN LET ymax = yset(point)
  261   REM  *   A useful mathematical technique there - to
find the largest number in a list
  270   NEXT point
  280 FOR point = 1 TO set
  290   LET xset(point) = INT(xset(point)*1280/xmax)
  300   LET yset(point) = INT(yset(point)*1024/ymax)
  301   REM  *   The scaling bit; see if you can suss it
  310   NEXT point
  320 ENDPROC
  330 DEF PROCdraw
  340 GCOL 0, 131: GCOL 0,1: CLG: VDU 5
  350 MOVE 0,32: PRINT "0": MOVE 0,1020: PRINT; ymax: MOVE 1
280-32*LEN STR$ xmax,32: PRINT; xmax
  351 REM  *   TAB won't work in grafix routines
  360 MOVE xset(1), yset(1)
  370 FOR point = 1 TO set-1
  380   PLOT 1, xset(point+1)-xset(point), yset(point+1)-yse
t(point)
  390   NEXT point
  400 ENDPROC
```

What this program does is to allow you to enter a set of point-pairs (in increasing orders of x, unless you want to cause great confusion!). It then re-scales the values of the coordinates of the points, so that the graph fits just onto the screen area, however small or large those values are. And then it plots them with simple axes.

PROJECTS

1. Make a thorough comparison of the readability of text, the contrast of colours, the cleanness of DRAWn lines, and so on, in the different modes. The best approach is to devise a simple program to run through the facilities. Take notes on your findings and refer to them during future program development.
2. Re-code Program 31 in MODE 2 to give lines in different colours at random, with each line drawn producing a SOUND. I'll leave it to you to decide the action for ESCAPE.
3. Develop Program 32 to give the hull (a combination of triangles in a chosen colour, and, maybe VDU 23 . . . characters) and background sea over part of the paper.
4. Use VDU 23, MOVE, DRAW and GCOL to program a MODE 5 screen display of a house. Now add animation of, for instance, birds flying around.
5. Try to develop an illustrated child-defined story program. It could start like this . . .

```
  50 PRINT' "Once upon a time there was a little boy/girl."
  60 PRINT''' "Well, you choose - Press B or G!"'''
  70 LET g$ = GET$
  80 IF g$<>"B" AND g$<>"G" THEN GOTO 70
  90 CLS
 100 REM  **  Draw a picture of the child
 110 PRINT' "What's the name of your little ";
 120 IF g$ = "G" THEN LET g$ = "girl" ELSE LET g$ = "boy"
 130 PRINT g$ "?"
 140 INPUT'' n$
 150 REM  **  Now add name to picture....
```

20 Hip Hip Array

Well, we *can* cheer. This is the last chapter of the book, and deals with the last remaining aspect of basic BASIC.

I've had to use its subject, *arrays* (the computer equivalent of lists and tables), in a number of programs already. That's because they are *so* useful. Really I can't think why they didn't get looked at in detail in an earlier chapter. Anyway . . .

I'd like to show you the power of the concept with a program called Bar-tender. This stores the favourite drinks of a number of named people. (Later versions will no doubt actually pour the drinks, but for the time being we'll have to leave that feature out.) Here's the program concept:

1. Ask the guest's name.
2. If the name is listed, offer choice of his or her three favourite drinks.
3. Otherwise, ask for those details and store them for later use.

This is what a real bar-tender does, mentally or otherwise:

Table 20.1

Guest	Drink 1	Drink 2	Drink 3
Mary	Small vodka	Large orange	Bitter
Mohammed	Iced lemon	Large tonic	Iced water
Margaret	Large neat whisky	Large neat whisky	Large neat whisky

and so on. He or she builds up a table of four columns and perhaps twenty or thirty lines (rows).

BASIC micros do the same. First, however, you must tell them the memory required so that this can be 'reserved' (an out-dated feature in fact). To do this we use a DIM (= dimension) statement. In this particular case, we have a *string array* of, say, thirty rows and four columns (thirty people, and their name and three favourite drinks). My program will start, therefore, with DIM drink$(3Ø, 4). In use, each guest's name (name$) must be tested against the first entry in each row until a match is found. Then we can go to a 'get drink' procedure. If there's no match, we'll need to go to a 'get order' procedure.

The loop is the obvious way to do the checking. It is straightforward because each *element* in an array carries a number (its *subscript*). Figure 20.1 shows the way the memory allocation looks (to humans). (In fact, in the Elk, the subscripts start at Ø, but this is not important to us.)

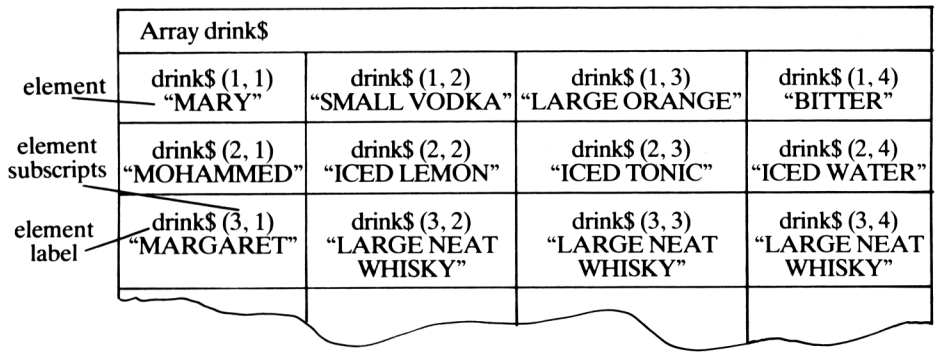

Array drink$
element
element subscripts
element label
drink$ (1, 1) "MARY"
drink$ (1, 2) "SMALL VODKA"
drink$ (1, 3) "LARGE ORANGE"
drink$ (1, 4) "BITTER"
drink$ (2, 1) "MOHAMMED"
drink$ (2, 2) "ICED LEMON"
drink$ (2, 3) "ICED TONIC"
drink$ (2, 4) "ICED WATER"
drink$ (3, 1) "MARGARET"
drink$ (3, 2) "LARGE NEAT WHISKY"
drink$ (3, 3) "LARGE NEAT WHISKY"
drink$ (3, 4) "LARGE NEAT WHISKY"

Figure 20.1

Here's the program. I'll go through it afterwards, as necessary, but check, and think on, the comments within as you enter it.

Program 34: Bar-tender

```
   10 DIM drink$(30,4)
   20 MODE 1: VDU 19 128 3 0;0 19 3 4 0;0: CLS
   30 PROCgetname
   40 PROCsearch
   50 LET g = GET: GOTO 20
   60 DEF PROCgetname
   70 PRINT'' "Hallo; my name is Georgina."''' "Please tell
 me YOUR name."'''
   80 INPUT name$: IF LEN name$ < 2 THEN GOTO 80
   81 REM  **  Mugtrap
   90 REPEAT
  100   CLS: PRINT''' "My eyes are bad."'' "Are you M(ale) o
r F(emale)?"'
  110   INPUT gender$
  120   UNTIL gender$ = "F" OR gender$ = "M"
  121 REM  **  Mugtrap
  130 CLS: PRINT'' "Thank-you, ";
  140 IF gender$ = "F" THEN LET tit$ = "Madam." ELSE LET tit
$ = "Sir."
  150 PRINT tit$
  160 ENDPROC
  170 DEF PROCsearch
  180 FOR element = 1 TO 30
  190    IF drink$(element,1)=name$ THEN LET flag = 0: PROCd
rink: ENDPROC
  200    IF drink$(element,1)="" THEN LET flag = 1: PROCorde
r: ENDPROC
  201   REM  **  Naughty to jump out of loops like this. Nau
ghty, but nice.
  210   NEXT element
  220 ENDPROC
  240 DEFPROCdrink
  250 IF flag = 0 THEN PRINT''' "Good to see you again.": LE
T wait = INKEY(200)
  260 PRINT''' "Here are your favourite drinks. Please  pres
s 1, 2, or 3 to choose!"
  270 FOR choice = 1 TO 3: PRINT' choice "  " drink$(element
,choice+1): NEXT choice
  280 LET g = GET-48: IF g<1 OR g>3 THEN GOTO 280
  281 REM  **  Mugtrap
  290 CLS: PRINT'' "Right, " tit$ "."'' drink$(element, g+1)
 " it is."'''' "I'll tell the waiter."''''': LET wait = INKEY
(50): SOUND 0,-15,0,20
  300 ENDPROC
  310 DEF PROCorder
```

```
  320 PRINT'' "I don't recall you."''' "Would you care to te
ll me your favouritedrinks?"
  330 LET drink$(element,1) = name$
  331 REM  **  First column of new entry
  340 FOR fav = 1 TO 3
  341   REM' **  Next three
  350   PRINT' "Number "; fav;: INPUT drink$(element,fav+1):
 IF LEN drink$(element,fav+1)<3 THEN GOTO 350
  360   NEXT fav
  370 FOR wait = 1 TO 1500: NEXT wait
  380 CLS: PROCdrink
  390 ENDPROC
```

We can see the two-dimensional drink$ array in four contexts here.

1. Being set up with DIM, line 10. Here I've allowed for 30 guests with three full choices each. The program will fail if more than 30 people are at your party, so modify this if you need. Use, for instance:

```
5  INPUT "How many guests do you fear you'll get", n
10 DIM drink$ (n, 4)
```

   However, note that once the program is running, you can't change the dimensions of an array. On the other hand, beware of making an array too big. Arrays use a lot of memory, and in sophisticated programs this can be a major problem. In our case Bar-tender won't work if you have more than about 400 guests, but then if you have parties that big you can probably afford a larger computer.

2. Having its elements entered, you can READ, LET, or INPUT. In Bar-tender, I used the last two, in lines 330–360. Here's the usual structure, illustrated with a one-dimensional numeric array. Of course a numeric array doesn't need a $ in its name.

```
   10 REM  **  RAINFALL  **
   20 REM  ****************
   30 INPUT'' "How many months",m
   40 DIM wet(m)
   50 FOR a = 1 TO m
   60   PRINT' "Rainfall for month "; a ":": INPUT SPC(25) w
et(a)
   70   NEXT a
```

3. Testing the elements of the array—lines 180–200 above. Again a loop is often used, as in this continuation of "Rainfall".

```
   80 FOR a = 1 TO m
   90   IF wet(a) > 1000 THEN PRINT "Query data for month ";
 a', "Please re-enter!": INPUT SPC(25) wet(a)
  100   NEXT a
```

4. Print out the array elements—line 270. Or with "Rainfall"—

```
  110 LET g = GET: CLS
  120 PRINT'' "Please check the data!"
  130 FOR a = 1 TO m
  140   PRINT' "Month ";a "  Rainfall: "; wet(a) " units"
  150   NEXT a
  160 REPEAT
  170   INPUT' "Any errors (Y/N)", check$
  180   IF LEFT$(check$,1) = "Y" THEN INPUT' "Month number",
a: INPUT "Rainfall", wet(a)
  190   UNTIL LEFT$(check$,1) = "N"
```

Apply all those ideas to Program 32, and to the next listing, another 'Bar' program . . .

Program 35: Bar chart

```
  10 VDU 23 223 170,85,170,85,170,85,170,85
  20 INPUT'' "Title", tit$
  30 INPUT'' "How many values (max 35)", n
  40 DIM a(n-1)
  41 REM  **  Takes account of element 0
  50 CLS
  60 FOR a = 1 TO n
  61   REM  **  No problem of overlapping variables
  70   PRINT' "Value number "; a; " ";:INPUT a(a-1)
  80   NEXT a
  90 MODE 1: COLOUR 130: COLOUR 0: CLS
 100 FOR c = 1 TO 30
 110   FOR b = 1 TO n
 120     IF a(b-1)<30-c THEN PRINT " "; ELSE PRINT CHR$ 223
;
 130     NEXT b: PRINT: NEXT c
 131 REM  **  Nested FOR loops
 140 FOR d = 1 TO n: PRINT RIGHT$(STR$ d,1);: NEXT d
 150 FOR d = 1 TO 30: PRINT TAB(39,d-1) RIGHT$(STR$(30-d),1
): NEXT d
 151 REM  **  Snazzy string-handling in last 2 lines
 160 PRINT TAB(2,5) tit$
 170 REPEAT: UNTIL 0
```

PROJECTS

1. Develop Bar-tender so that Georgina offers known guests the chance to change their list. Also add suitable colour and sound routines.
2. Devise a simple directory program to hold names and 'phone numbers in *two* string arrays, so that you can get the 'phone number of any friend listed by entering the name. Start off with

   10 DIM name$ (20) number$ (20)

   if there are twenty records.
3. Use a similar parallel pair of one-dimension arrays to hold the names and game scores of, say, thirty players. Start with:

   10 DIM name$ (30) score (30)

   Try to set it up so that you have a menu to choose from, to give such things as highest score, lowest score and mean score (average); full list in name order; full list in score order. Use procedures for all these tasks.

Appendix 1: Additional Programs

Program A1(a): Printout

This is a very simple (but still, I hope, useful) program to introduce you to some major programming ideas early on. Enter it, check it, RUN it, correct it (if need be), and think about it. Once the program is running properly, you will need to press the ESCAPE button to stop it.

```
 10 MODE 5
 20 COLOUR 2
 30 COLOUR 129
 40 CLS
 50 REPEAT
 60   PRINT'' "Give me a number!"''
 70   INPUT number
 80   LET square = number * number
 90   LET cube = square * number
100   PRINT " Number: " number
110   PRINT " Square: " square
120   PRINT " Cube:   " cube
130   UNTIL FALSE
```

Program A1(b): Bonus

As the above will seem very trivial once you've got far into this book, here's a bonus program which I hope you'll enjoy somewhat longer! It's a simple pattern-generator, the pattern being made of repetitions of the string you input in response to the first line. So try to input 'messages' consisting of nice pattern-generating symbols . . . .

```
 10 INPUT "Your pattern base", p$
 20 CLS
 30 MODE 2
 40 REPEAT
 50   FOR col = 0 TO 15
 60     COLOUR col+128
 70     FOR print = 1 TO 640/(LEN p$ + 1)
 80       REPEAT
 90         LET ink = RND(16)-1
100         UNTIL ink <> col
110       COLOUR ink
120       PRINT p$ " ";
130       NEXT print
140     SOUND 1, -15, col*15, 20
150     NEXT col
160   VDU 7
170   MODE 6
180   RUN
```

Program A2: Train-race

No way is this a novel game—two engines race across the screen, moving forward a random distance each move until one or other reaches the end. The best you can do with it is to bet on the winner.

The principle is the basis of a number of similar exercises, non-interactive or interactive. It is straightforward to add twiddly bits like reverse as well as forward motion, barriers, and switching tracks (one reason I chose MODE 6, which gives you the 'railway lines' for free).

```
 10 VDU 23 225 0,255,65,73,77,75,127,127
 20 VDU 23 226 0,0,0,0,0,24,255,255
 30 VDU 23 227 0,28,28,28,28,28,254,254
 40 VDU 23 228 127,127,127,127,255,13,7,2
 50 VDU 23 229 255,255,255,255,255,155,14,4
 60 VDU 23 230 254,254,254,254,255,54,28,8
 70 VDU 23 231 16,168,84,170,84,120,20,40
 80 MODE 6
 90 VDU 5
100 VDU 19 1,0,0;0 19 0,3,0;0
110 REPEAT: CLS
120   LET a1 = 0: LET a2 = 0
130   PRINT TAB(36,10) "Me" TAB(36,16) "You"
140   REPEAT
150     LET a1 = a1+RND(3)
160     PROCtrain(a1,9)
170     LET a2 = a2+RND(3)
180     PROCtrain(a2,15)
190     UNTIL a1>26 OR a2>26
200   PRINT TAB(2,20);
210   IF a1>26 THEN PRINT "My"; ELSE PRINT "Your";
220   PRINT " train won!"
230   FOR w = 1 TO 2000: NEXT w
240   UNTIL FALSE
250 DEF PROCtrain(x,y)
260 PRINT TAB(x+4,y) CHR$ 231: SOUND 1,-15,y*4,4
270 PRINT TAB(x,y+1) "   "+CHR$ 225+CHR$ 226+CHR$ 227 TAB(
x,y+2) "   "+CHR$ 228+CHR$ 229+CHR$ 230
280 FOR w = 1 TO 1500: NEXT w
290 ENDPROC
```

Program A3: Sketch pad

This program lets you build up a 'picture' using any keyboard character within the MODE 5 'canvas'. You can make it as sophisticated as you wish, but already the user instructions need thought. Here they are:

1. If an inverse "?" is on screen, whatever character you enter except "S" will appear at that position. If you enter "S" the program stops.
2. If there is no inverse "?" enter "U", "D", "L", or "R" to move the 'cursor' up, down, left or right (unless it's at the edge of the screen).

Can you modify this to allow the input of DIY block characters? I've put some ideas in Appendix 2.

```
 10 MODE 1: COLOUR 130: COLOUR 0: CLS
 20 VDU 23 225 193,221,253,225,239,239,255,239
 30 LET x = 20: LET y = 15
 40 PRINT TAB(x,y) CHR$ 225
 50 REPEAT: LET a$ = GET$: UNTIL a$<>""
 60 IF a$ = "S" THEN REPEAT: UNTIL FALSE
 70 PRINT TAB(x,y) a$
 80 REPEAT: LET a$ = GET$: UNTIL a$<>""
 90 IF a$="U" AND y>2 THEN LET y = y-1: GOTO 40
100 IF a$="D" AND y<29 THEN LET y = y+1: GOTO 40
```

```
110 IF a$="L" AND x>2 THEN LET x = x-1: GOTO 40
120 IF a$="R" AND x<38 THEN LET x = x+1: GOTO 40
130 GOTO 80
```

Program A4: Wallpaper

This is an impress-your-visitors-with-your-new-computer type of program. Not deep, but it includes some nice effects.

```
 10 MODE 2: COLOUR 2: COLOUR 133: CLS
 20 PRINT TAB(2,3) "What is your name";: INPUT name$
 30 COLOUR 1: COLOUR 131: CLS
 40 PRINT TAB(2,3) "Thanks, " name$ "."
 50 LET g = INKEY 250
 60 VDU 19 1 6 0;0
 70 REPEAT
 80   PRINT'' " Press a number key!"
 90   *FX 15,0
100   LET b = INKEY(400)-48: PROCchime
110   UNTIL b = 0
120 CLS
130 VDU 19 131 4 0;0
140 FOR a = 1 TO 300: PRINT name$ " ";: NEXT a
150 LET c = 0
160 REPEAT
170   VDU 19 131 c 0;0
180   FOR a = 1 TO 100
190     GCOL RND(15),RND(15)
200     PLOT 85, RND(1279), RND(1023)
210     SOUND 1,-15,a,a/5
220     FOR b = 1 TO 100: NEXT b
230     NEXT a
240   LET c = c + 1
250   UNTIL c = 16
260 RUN
270 DEF PROCchime
280 FOR c = 1 TO b
290   VDU 7: LET i = INKEY 50
300   NEXT c: LET i = INKEY 20
310 LET i = INKEY 200
320 ENDPROC
```

Program A5: Reflex timer

No notes needed here, I think.

```
  10 MODE 5: COLOUR 130: COLOUR 0: REPEAT: CLS
  20   PRINT TAB(63) "REFLEX TIMER"' "   ============"'''''
 "Enter number of yourchoice:"'''' " 1. Simple - screen"'''
" 2. Harder - sound"''' " 3. Harder - both"'''''''
  30   LET lvl = ASC(INKEY$(3E4))-48: IF lvl<1 OR lvl>3 THE
N GOTO 30
  40   CLS: LET tb = 1000: LET sum = 0: ON lvl GOSUB 50, 12
0,190: UNTIL FALSE
  50 FOR q = 1 TO 10: CLS: PRINT''' "TEST 1"''' "GO   "; q'
''
  60   TIME = 0: REPEAT: UNTIL TIME>100
  70   PRINT'''' "Press RETURN key""when screen blanks!"''
'
  80   PROCtime1: CLS
  90   PROCtime2
 100   NEXT q
 110 RETURN
 120  FOR q = 1 TO 10: CLS: PRINT''' "TEST 2"''' "GO   "; q
'''
 130   TIME = 0: REPEAT: UNTIL TIME>100
```

```
  140   PRINT''' "Press RETURN key"'"when buzzer buzzes!"''

  150   PROCtime1: VDU 7
  160   PROCtime2
  170   NEXT q
  180 RETURN
  190 FOR q = 1 TO 10: CLS: PRINT''' "TEST 3"''' "GO   "; q'
''
  200   TIME = 0: REPEAT: UNTIL TIME>100
  210   PRINT''' "Press RETURN key"'"when buzzer buzzes"'"a
fter screen clears!"'''
  220   PROCtime1: CLS: PROCtime1: VDU 7
  230   PROCtime2
  240   NEXT q
  250 RETURN
  260 DEF PROCtime1
  270 FOR b = 1 TO 500+RND(2500): NEXT b
  280 ENDPROC
  290 DEF PROCtime2
  300 TIME=10: *FX 15,1
  310 LET i = INKEY 3E4: LET t = TIME/100
  320 LET sum = sum+t: LET tm = sum/q: IF t<tb THEN LET tb =
 t
  330 CLS: PROCresult
  340 PRINT''' "RETURN to go on."''': INPUT b: ENDPROC
  350 DEF PROCresult
  360 PRINT''' "Result: "; t "sec"
  370 PRINT''' "Best:   "; tb "sec"
  380 PRINT''' "Mean:   "; INT(100*tm+0.5)/100 "sec"
  390 PRINT''' "Go "; q
  400 ENDPROC
```

Program A6: Guess the hard number

This is a much posher version of Program 1 (Page 13) with three levels of difficulty, and as a result, just about the most complex numerical expression I've ever written (line 80). That uses logical sub-expressions as factors. The rest is fairly straightforward to make up for that!

```
   10 MODE 5: COLOUR 3: COLOUR 129: CLS
   20 LET rec = 100: LET limit = 99
   30 REPEAT
   40   CLS: LET go = 0: PRINT TAB(24) "NUMBER GUESS"' TAB(4
) "_______ ______" TAB(0,10) "Please type your"'' "name & RETU
RN!"''''
   50   INPUT name$: PRINT'' "Thank-you, " name$ "."
   60   REPEAT: INPUT'' "Level (1, 2 OR 3)", L: UNTIL L>0 AN
D L<4 AND L=INT L
   70   LET wait = INKEY(250): CLS
   80   LET z = RND(1): LET tgt = (RND(limit)*(-(L=1)))+((1+
(INT(RND(1)*limit*10))/10)*(-(L=2)))+((1+(INT(RND(1)*limit*1
0))/10)*((-1*(z>0.5)+(z<=0.5))*(L=3)))
   90   PRINT'' "I am thinking of a  number up to "; limit+1
 "."
  100   REPEAT
  110     INPUT' "Please enter guess."' guess: LET go = go +
 1
  120     IF guess<tgt THEN PRINT "Too LOW..." ELSE IF guess
>tgt THEN PRINT "Too HIGH..." ELSE PRINT "RIGHT!!"
  130     UNTIL guess = tgt
  140   PRINT''' "Got it in "; go "!"
  150   IF go<=rec THEN LET rec = go: LET rec$ = name$: LET
reclvl = L: PRINT'' " A RECORD!!!"
  160   LET wait = INKEY 400: CLS
  170   PRINT''' "Best score so far: ", rec'' "scored by "
rec$' "Level ";reclvl''''
  180   LET wait = INKEY 250
  190   PRINT'' "Press RETURN to try again."'': INPUT s$
  200   UNTIL FALSE
```

Program A7: Hangman

```
  10 VDU 23 224 255,255,255,255,255,255,255,255
  20 VDU 23 225 255,255,255,255,129,255,255,255
  30 VDU 23 226 13,11,13,11,13,11,13,11
  40 VDU 23 227 176,208,176,208,176,208,176,208
  50 VDU 23 228 0,0,1,3,3,3,7,7
  60 VDU 23 229 0,170,255,221,136,255,247,247
  70 VDU 23 230 0,128,192,224,224,224,240,240
  80 VDU 23 231 7,7,3,1,1,1,0,0
  90 VDU 23 232 227,255,255,193,193,255,255,255
 100 VDU 23 233 240,240,224,192,192,192,128,0
 110 MODE 5: COLOUR 129: COLOUR 0: *FX 202,16
 111 REM * CAPS LocK off
 120 CLS: FOR q = 1 TO 6
 130   READ q$: LET l = LEN q$: LET ok = 0: LET no = 0: LET
go = 0: LET u$ = " "
 140   PROCdash
 150   REPEAT
 160     PRINT TAB(2,1) "Used:" u$ TAB(2,27) "Enter letter!
"
 170      REPEAT: LET g = GET: PROCtest: UNTIL f = 0: REM *
Mug-traps
 180     LET go = go+1: PROCcheck
 190     UNTIL ok = l OR no = 8: REM * Succeed or fail
 200   PRINT TAB(2,27) STRING$(13," "): TIME = 0:  REPEAT:
UNTIL TIME>750: CLS: NEXT q: VDU 19 0 15 0;0:PRINT TAB(4,15)
 "That's all!"''''': REPEAT: UNTIL 0
 210 DATA strength,weakness,program,hangperson,elk,byebye
 220 DEF PROCdash
 230 PRINT TAB(2,10) STRING$(l,CHR$ 225): ENDPROC
 240 DEF PROCtest
 250 LET g$ = CHR$ g: LET f = 0: REM Flag f put off
 260 IF g<97 OR g>122 OR INSTR(u$,g$)>0 THEN LET f = 1: END
PROC
 270 LET u$ = u$ + g$: ENDPROC
 280 DEF PROCcheck
 290  LET f = 0: FOR a = 1 TO l: IF g$ = MID$(q$,a,1) THEN
PROCok: REM * Good guess
 300   NEXT a: IF f = 0 THEN PROCno: REM * Bad one
 310 ENDPROC
 320 DEF PROCok
 330 LET ok = ok+1: LET f = 1: PRINT TAB(a+1,10) g$:ENDPROC
 340 DEF PROCno
 350 LET no = no+1: GOSUB(1000+no*50): ENDPROC:   Computed
sub-routine address
1050 PRINT TAB(10,25) STRING$(9,CHR$ 224): RETURN
1100 FOR c = 24 TO 2 STEP -1: PRINT TAB(14,c) CHR$ 224:NEXT
c: RETURN
1150 FOR c = 14 TO 10 STEP -1: PRINT TAB(c,2) CHR$ 224:NEXT
c: RETURN
1200 FOR c = 3 TO 13: PRINT TAB(11,c) ".": NEXT c: RETURN
1250 FOR c = 21 TO 24:PRINT TAB(10,c) CHR$ 226+" "+CHR$ 227
: NEXT c: RETURN
1300 FOR c = 18 TO 20: PRINT TAB(9,c) CHR$ 226+CHR$ 226+CHR
$ 224+CHR$ 227+CHR$ 227: NEXT c: PRINT TAB(9,17) CHR$ 226+ST
RING$(3,CHR$ 224)+CHR$ 227: RETURN
1350 PRINT  TAB(10,14) CHR$ 228+CHR$ 229+ CHR$ 230+CHR$ 10+
STRING$(3,CHR$ 8)+CHR$ 231+CHR$ 232+CHR$ 233 TAB(11,16) CHR$
 224: RETURN
1400 PRINT TAB(2,28) "HANGED      " TAB(2,10) q$: RETURN
```

This got longer than I hoped—and better than I expected. You may wish to improve the user-defined graphics—but I've seen far less realistic hangmen programs! (Don't think I'm a violent person on the basis of this, though.)

Program A8: Tables test

```
   10 VDU 23 224 255,170,170,170,170,170,128,193
   20 VDU 23 225 247,247,247,247,247,247,247,255
   30 VDU 23 226 255,247,231,231,231,231,231,231
   40 LET menu$ = CHR$ 224 + CHR$ 226 + CHR$ 10 + CHR$ 8 + C
HR$ 8 + CHR$ 225 + CHR$ 225
   50 REPEAT
   60   MODE 5: COLOUR 129: COLOUR 0
   70   CLS: PRINT TAB(3,2) menu$ TAB(7,3) "MENU" TAB(7,4) "
====" TAB(13,2) menu$
   80   PRINT''' "1. Show table and   test"'' "2. Test, no s
how"'' "3. Test on 2-5"'' "4. Test on 6-12"'' "5. Test on 2-
12"'''
   90   FOR wt = 1 TO 1500: NEXT wt
  100   PRINT "Enter number of yourchoice."'''
  110   REPEAT: * FX 15 0
  111     REM  **  mpties input 'buffer'; no REM allowed aft
er any * instruction
  120     LET lev = GET-48
  130     UNTIL lev > 0 AND lev < 6: REM  **  Full mug-trapp
ing
  140   LET f1 = -(lev=1): LET f2 = -(lev=2): LET f3 = -(lev
=3): LET f4 = -(lev=4): LET f5 = -(lev=5): REM ** Sets flags
 using logical expressions
  150   IF f1 OR f2 THEN INPUT'' "Which table", tab: IF tab
< 2 OR tab > 12 THEN GOTO 150: REM  **  REPEAT...UNTIL as in
 110-130 is better!
  160   IF f1 THEN PROCtable(tab)
  170   LET score = 0
  180   FOR ques = 1 TO 10
  190     LET mlt = 1 + RND(11): IF f3 THEN LET tab = 1 + RN
D(4) ELSE IF f4 THEN LET tab = 5 + RND(7) ELSE IF f5 THEN LE
T tab = 1 + RND(11): REM  **  Chooses question data
  200     PRINT'' "Question "; ques' "What is "; mlt; " time
s "; tab; "?",: INPUT ans
  210     IF ans = mlt*tab THEN PRINT, "Yes!": LET score = s
core + 1 ELSE PRINT, "No!"
  220     NEXT ques
  230   VDU 19 0 4 0;0; REM  **  Change ink colour
  240   PRINT''' "Your score is "; score; ".": FOR wt = 1 TO
 5000: NEXT wt
  250   UNTIL FALSE: REM  **  Go on for ever and ever....
  500 DEF PROCtable(t)
  510 FOR line = 1 TO 12: REM  **  Or change to 10 for moder
n contexts
  520   PRINT' ;: IF line < 10 THEN PRINT " ";
  530   PRINT; line; " times "; t; " is "; line*t
  540   NEXT line: ENDPROC
```

A sophisticated menu-driven version of a standard 'teaching' program. Hope you like the menu!

Program A9: Target practice

```
   10 VDU 23 225 3,6,60,40,104,60,126,255
   20 VDU 23 226 36,90,165,90,60,155,24,60
   30 VDU 23 227 24,36,66,153,153,66,36,127
   40 VDU 23 228 28,42,85,170,127,170,85,255
   50 MODE 5: GCOL 0,130: GCOL 0,0: COLOUR 130: COLOUR 1
   60 REPEAT: CLG: LET a = RND(4)-1: LET b = RND(4)+15: LET
c = RND(9)+1: LET d = 0
   70   REPEAT: LET d = d + 1: PROCsetup
   80     INPUT TAB(1,1) "Angle ",a1: INPUT TAB(1,2) "Speed
", e
   90     LET an = a1*PI/180: LET x = 8
  100     REPEAT
  110       LET x1 = x + 64*(a+1)
  120       LET y = 32 + x*TAN(an) - x^2/(e^2*COS(an)^2)
```

```
  130        PLOT 69,x1,y+32: SOUND 1,-15,y/5+20,2
  140        LET x = x + 16
  150        UNTIL x>1280 OR y<=32 OR ABS POINT(x1+9,y+32)=1:
 REM * Crash
  160      PRINT TAB((x1+32)/64,31-y/32) CHR$ 226:SOUND 0,-15
,6,20: FOR w = 1 TO 3000: NEXT w
  170      CLG: UNTIL d = 10 OR ABS(x1-b*64)<40: REM * Fail
or succeed
  180    IF d = 10 THEN PRINT TAB(6,10) "Useless!"''''' ELSE
PRINT TAB(4,10) "Good shot!"'' "  You got it in "; d "."
  190    FOR w = 1 TO 2500: NEXT w: PRINT''' "Have another go
...": FOR w = 1 TO 1000: NEXT w: UNTIL 0
  200 DEF PROCsetup: LOCAL d
  210 PRINT TAB(a,31) CHR$ 225 TAB(b,31) CHR$ 227
  220 FOR d = 1 TO c:PRINT TAB(5,30-d+1) STRING$(c,CHR$ 228)
: NEXT d: ENDPROC
```

In each run you have ten attempts to hit the target with a shot from a gun the other side of a barrier. The routine in lines 120–150 plots the parabolic trajectory (you could add lines to bring in air friction and wind if you wish). For each run, gun and target are placed at random and the barrier has random height and width.

Program A10: Guess the word

```
   10 MODE 5: COLOUR 130: COLOUR 1
   20 VDU 23 224 85,170,85,170,85,170,85,170
   30 VDU 23 225 255,255,255,255,129,255,255,255: REM *
 Inverse -
   40 VDU 23 226 251,247,239,223,191,223,239,247: REM *
 Inverse <
   50 VDU 23 227 191,223,239,247,251,247,239,223: REM *
 Inverse >
   60 LET b$ = CHR$ 224: LET title$ = STRING$(16,b$)+CHR$ 10
+STRING$(16,CHR$ 8)+b$+"GUESS THE WORD"+b$+CHR$ 10+STRING$(1
6,CHR$ 8)+STRING$(16,b$): LET c$ = STRING$(20," ")
   70 LET word = 0
   80 REPEAT
   90   LET word = word+1
  100   LET go = 0: CLS
  110   PRINT TAB(22) title$:  READ target$:   LET len = LEN
  target$
  120   FOR ch = 1 TO len: PRINT TAB(ch+2,10) CHR$ 225: NEXT
 ch
  130   REPEAT
  140     LET check=0: PRINT TAB(0,14) "Enter guess/RETURN."
:    INPUT guess$: LET go = go+1
  150     IF LEN guess$ <> len  THEN  PRINT "Wrong length:
AGAIN!": GOTO 140
  160      FOR ch = 1 TO len: IF MID$(guess$,ch,1)<MID$(targ
et$,ch,1) THEN PRINT TAB(ch+2,10) CHR$ 227 ELSE IF MID$(gues
s$,ch,1)>MID$(target$,ch,1) THEN PRINT TAB(ch+2,10) CHR$ 226
 ELSE PRINT TAB(ch+2,10) MID$(target$,ch,1): LET check = che
ck+1
  170       NEXT ch: PRINT TAB(0,15) c$,c$
  180     UNTIL check = len
  190   PRINT''' c$''' "Got it - "; go " goes."''''
  200   IF word = 6 THEN  FOR w = 1 TO 2500:  NEXT w:  CLS:
PRINT TAB(2,12) "Well, time's up!"''''': END
  210   INPUT TAB(2,30) "Again (Y or N)", y$
  220   UNTIL y$<>"Y" AND y$<>"y"
  230 CLS: PRINT TAB(2,12) "That's it, then!"''''''': END
  240 DATA COMPUTER, ACORN, WORD, BUZZ, GUESS, QUARK
```

This is rather an intriguing word game that requires a fair amount of thought. At each attempt, you must enter a word of the right length (the display shows how many letters there are). The computer's response is, as one may expect, to replace dashes by correct letters in the correct places. For an incorrect letter, the display turns to an inverse '>' if

the guess is earlier in the alphabet than the correct letter; inverse '<' indicates that your letter is later in the alphabet than it should be. The DATA in line 240 may of course be altered at will.

Program A11: Lunarvader

```
  10 VDU23 224 4,7,15,31,57,57,63,126
  20 VDU23 225 32,224,240,248,156,156,252,126
  30 VDU23 226 255,63,39,37,36,36,36,108
  40 VDU23 227 255,252,228,164,36,36,36,54
  50 VDU23 228 1,1,1,1,1,1,1,1
  60 CLS: INPUT'' "What level (0:easy-19:awful)", lvl
  61 REM  **  YOU can mug-trap this!
  70 PRINT TAB(5,10); "Z: move left; /: move right"
  80 LET x$ = INKEY$(500): REM  * Delay
  90 REM  *  Start of actual game....
 100 REPEAT: *FX 15
 101   REM  *  Clear input buffer; NB no statement after *F
X,  not even a REM
 110   MODE 5: GCOL 0,130: VDU 5: GCOL 0,1: CLG
 120   LET s = RND(1000+lvl*5)+100: PROCpin
 130   LET x = RND(1150+lvl*5): LET y = 1000: LET a = 1
 140   REPEAT: CLS: PROCpin: PROCc
 150     LET x$ = INKEY$(96-lvl*5): REM  *  Varies speed
 160     IF x$ = "Z" AND x > 50-lvl*5 THEN LET x = x-(15-a)
*10
 170     IF x$ = "/" AND x < 1200+lvl*5 THEN LET x = x+(15-
a)*10
 180     LET a = a+1: LET y = y-70
 190     UNTIL a = 15: REM  *  End go-loop
 200   MOVE 100,600: IF ABS(s-x)<7 THEN PRINT "SPLAT" ELSE
IF ABS(s-x)<15 THEN PRINT "Keep trying" ELSE PRINT "USELESS!
"
 210   FOR dly = 1 TO 2000-lvl*100: NEXT dly: UNTIL FALSE:
REM  *  Next go
1000 DEFPROCpin
1010 MOVE s,32: PRINT CHR$ 228 + CHR$ 8 + CHR$ 10 + CHR$ 22
8
1020 ENDPROC
1050 DEFPROCc
1060 MOVE x,y: PRINT CHR$ 224 + CHR$ 225
1070 MOVE x,y-32: PRINT CHR$ 226 + CHR$ 227
1080 SOUND 0,-15,a*15,a
1090 ENDPROC
```

As the name indicates, this is a cross between 'lunar lander' and 'space invader'. I'll leave you to work out what to do . . .

Program A12: Pie chart

```
  10 MODE 6: CLS: PRINT'' "PIE-CHART" ' "=========""''''
  20 INPUT "How many slices", seg
  30 DIM seg(seg): LET total = 0
  40 FOR slice = 1 TO seg
  50   PRINT' "Value "; slice;: INPUT seg(slice)
  60   LET total = total+seg(slice)
  70   NEXT slice
  80 LET inc = 2*PI/total
  90 LET a0 = 0: LET x0 = 600: LET y0 = 500: LET rad = 300
 100 MODE 0: VDU 19 0,3;0;: VDU 19 1,4;0;: CLG: VDU 5
 110 PROCcircle(x0,y0)
 120 FOR slice = 1 TO seg
 130   LET angle = inc*seg(slice): LET dx = rad*SIN a0: LET
dy = rad*COS a0
 140   SOUND 1,-15,1,1
 150   MOVE x0,y0: DRAW x0+dx, y0+dy
 160   LET a0 = a0+angle
```

```
  170   NEXT slice
  180 LET wait = GET: RUN
 1000 DEFPROCcircle(x,y)
 1010 LET t = 0
 1020 MOVE x+rad,y
 1030 FOR point = 1 TO 501
 1040   DRAW x+rad*COS t, y+rad*SIN t
 1050   LET t = t+PI/250
 1060   NEXT point
 1070 ENDPROC
```

This is just to be serious for a change—a fairly unsophisticated version of what, for some people, is an important routine. You may like to develop it to incorporate shading and labels, but this version works pretty well. Note the circle-drawing procedure.

Program A13: The Cambridge plate

```
   10 DIM h(5): DIM dh(5)
   20 VDU 23, 224, 0,0,0,0,0,1,3,63
   30 VDU 23, 225, 48,48,36,119,207,132,236,252
   40 VDU 23, 226, 127,191,191,185,81,81,80,80
   50 VDU 23, 227, 252,252,248,40,40,168,40,40
   60 MODE 4: VDU 19, 1,0,0,0,0: VDU 19, 0,2,0,0,0: CLS
   70 FOR line = 10 TO 30
   80   PRINT TAB(3, line); "¦"; TAB(36, line); "¦"
   90   NEXT line
  100 LET h(1) = 12: LET h(2) = 16: LET h(3) = 20: LET h(4)
= 24: LET h(5) = 28
  110 LET dh(1) = 0: LET dh(2) = 0: LET dh(3) = 0: LET dh(4)
 = 0: LET dh(5) = 0
  120 PROChorse
  130 PRINT TAB(38,12); 1; TAB(38,16); 2; TAB(38,20); 3; TAB
(38,24); 4; TAB(38,28); 5
  140 LET wait = INKEY(200): PRINT TAB(2,2); "Place your bet
s and press ENTER!"''
  150 LET g = GET
  160 SOUND 1,-15,53,40:SOUND 2,0,0,10: SOUND 3,0,0,20: SOUN
D 0,0,0,50: SOUND 2,-15,77,30: SOUND 3,-15,89,20: SOUND 0,-1
5,4,2
  170 LET wait = INKEY(200): PRINT TAB(2,2); "THEY'RE OFF!!
                  "''
  180 LET wait = INKEY(50)
  190 REPEAT
  200   LET dh(1) = dh(1) + INT(RND(1)+.5): LET dh(2) = dh(2
) + INT(RND(1)+.5): LET dh(3) = dh(3) + INT(RND(1)+.5): LET
dh(4) = dh(4) + INT(RND(1)+.5): LET dh(5) = dh(5) + INT(RND(
1)+.5)
  210   PROChorse
  220   IF dh(1) = 5 THEN PRINT TAB(2,2); "                "
  230   UNTIL dh(1) = 34 OR dh(2) = 34 OR dh(3) = 34 OR dh(4
) = 34 OR dh(5) = 34
  240 FOR horse = 1 TO 5: IF dh(horse) >33 THEN winner = hor
se
  250   NEXT horse
  260 VDU 19, 1,13,0,0,0
  270 LET wait = INKEY(400): VDU 19, 1,4,0,0,0
  280 PRINT TAB(2,2); "And the winner is horse "; winner; "!
"
  290 LET wait = INKEY(300): RUN
  300 DEF PROChorse
  310 FOR horse = 1 TO 5
  320   SOUND horse,-15,horse*10+10,3
  330   PRINT TAB(dh(horse),h(horse)); " ";CHR$ 224; CHR$ 22
5; TAB(dh(horse),h(horse)+1); " "; CHR$ 226; CHR$ 227
  340   LET wait = INKEY(5)
  350   NEXT horse
  360 ENDPROC
```

Like Program A2, this is a non-interactive game with which you can do no more than bet on the outcome. It is a race between five, er, horses. As with A2, however, there is no difficulty in making it into a sort of educational drill, with questions in DATA lines to answer and forward movement of a given horse only if that player gets the answer correct.

Program A14: Snake in the grass

So many games routines involve something bouncing around within a box, that this was one of the first things I tried working on when I got my BBC computer a couple of years before the Electron. I wanted to have a 'snake' (a short line) moving in that kind of way; I got there, but the route was far more interesting than the result, and I am therefore taking the liberty of reproducing it in Electron form.

Try these routines, they should teach you a great deal about this kind of programming, as they did me.
First we initialize:

```
10  MODE 5
20  GCOL 0, 130          (background to be yellow)
30  VDU 5                (switch off cursor)
40  CLG                  (present that background)
50  GCOL 0, 0            (foreground to be black)
```

The first exercise involves drawing a line:

```
60  LET x = 30: LET y = 25          (define start point)
70  MOVE x, y                       (go there)
80  LET xi = 6: LET yi = 6          (define increments—try
                                     changing them)
90  LET x = x + xi: LET y = y + yi  (step along line . . .)
100 PLOT 69, x, y                   (. . . plotting foreground point . . .)
130 GOTO 90                         (. . . at each step)
```

Then we introduce the 'bounce' idea. This means changing the sign of the 'increment' each time the line reaches an edge of the box.

```
110 IF x > 1250 OR x < 30 THEN LET xi = -xi: SOUND 1, -15, y/5, 10
120 IF y > 1000 OR y < 25 THEN LET yi = -yi: SOUND 2, -15, x/5, 10
```

The result is an utterly boring yet fascinating routine in its own right. One can spend hours in it when trying such variations as these:

1. Different colours (lines 20 and 50, and use of the VDU 19 . . . statement).
2. Different, and differing, increments (line 80).
3. Different boundaries (lines 110 and 120).

No two patterns are alike; many are remarkably hypnotic.

However, games involve bouncing balls rather than spaghetti junctions of lines. (Program 18 on Page 81 is more like it.) The difference is that one must erase the trail behind the head of the line:

```
55  LET a = 1
105 PLOT 71, x - xi * a, y - yi * a
```

(PLOT 71 means plot a point out to the coordinates marked, in the background colour.)

You should now be able to develop an Elk "breakout". However, *I* still have my snakes to think of. Increasing the value of 'a' (line 55) provides one. It also generates the problem of grass growing under the snake's feet.

Just as well use that, to obtain my last hypnotic routine. Keep line 55 as it is, and edit lines 11Ø and 12Ø by inserting "LET a = a + 1" at the end of each.

Generating pretty patterns is just as much fun as writing games. To get rid of the grass is another story. It involves setting up a separate set of coordinates and increments for the PLOT 71 to operate on. Try it, add some sound and some interaction, and there's a nice game for you!

Program A15: Big print

```
 10 MODE 6
 20 VDU 23 225 85,170,85,170,85,170,85,170
 30 REPEAT
 40   CLS: READ disp$: IF disp$ = "END" THEN RUN
 50   PROCblank: REM  *  Fills string with spaces to 15 characters
 60   FOR line = 1 TO 17 STEP 8
 70     LET line$ = MID$(disp$, line/8*5+1,5)
 80     PROCmag
 90     LET wait = GET
100     NEXT line
110   UNTIL Ø

200 DATA "10000/ 250=  40"
210 DATA "   20x  25= 500"
220 DATA "   36+  63=  99"
230 DATA "END"
1000 DEF PROCblank
1010 IF LEN disp$ < 15 THEN LET disp$ = disp$+STRING$(15-LEN disp$," ")
1020 ENDPROC

1050 DEF PROCmag
1060 LOCAL x,y
1070 FOR y = Ø TO 7
1080   PRINT TAB(Ø, line+y);
1090   FOR char = 1 TO 5
1100     LET plot$ = MID$(line$, char, 1)
1110     LET code = ?((ASC(plot$)-32)*8 + &CØØØ + y)
1120     FOR x = 7 TO Ø STEP -1
1130       IF (code AND (2^x)) = 2^x THEN PRINT CHR$ 225; ELSE PRINT " ";
1140       NEXT x: NEXT char: NEXT y
1150 ENDPROC
```

As I am a teacher, I hope you'll forgive my making the data in the last program of the book specifically suited to teachers. As it stands, it stores a set of sums in DATA lines (12Ø on); each sum is then displayed as three lines in big print characters, line by line (PROCmag). You should be able to adapt it to your own needs, including scrolling. However in the mode I here use, MODE 6, you are restricted to five characters a line.

Appendix 2: Notes on Questions

Please note that, unlike arithmetic, there is always more than one correct answer to a computer programming exercise. The first criterion is—does your solution work? If it does, you should then try to assess its speed and memory demand compared to another answer.

Here is a load of leads to answers, for some of the questions I've posed anyway.

CHAPTER 3

Program 1 (amended)

```
   10 MODE 5
   12 REPEAT
   14   LET foreground = RND(4)-1
   16   LET background = RND(4)-1
   18   UNTIL foreground <> background
   20 COLOUR background + 128
   25 COLOUR foreground
   30 CLS
   40 LET unknown = RND(10)
   50 REPEAT
   60   PRINT "Please give me a    number between 1    and 1
0 and then     press ENTER."
   70   INPUT answer
   72   IF answer = 0 THEN STOP
   75   IF answer <> unknown THEN PRINT "Sorry - try again!"
   80   UNTIL answer = unknown
   90 PRINT"***  Right!  ***      "
  100 PRINT "Press RETURN and youget another go...."
  110 INPUT go
  120 RUN
```

1. (a) See lines 72 and 100–120 (you'll find far better ways later).
   (b) See lines 12 to 25. ('Why not?' If foreground and background colours are the same, you've a readability problem on your hands, I mean screen.)
   (c) See line 75.
   (d) See lines 60, 90 and 100—I hope you agree, these are a bit more friendly. Now you can see why we space line numbers in tens!

   NOTE that this is *not* the best way to develop a program. As we'll discuss in Chapter 14 in particular, programs should be planned fully before you touch the keyboard. I'm sure that you, viewing this as a user, can think of imperfections such as crowded layout. Fear not—with a few more hours' work you'll be able to make it really snazzy.

2. You could use, for instance,

```
76  IF answer < unknown THEN PRINT "Go up."
78  IF answer > unknown THEN PRINT "Go down."
```

3. Here's one way

```
45  LET guess = Ø
71  LET guess = guess + 1
95  PRINT "You had □"; guess; "□ goes."
```

NOTE that if you have entered all these lines, the listing will be awfully messy. You can improve that using the command RENUMBER(R) before listing again.

CHAPTER 5

4. (a) These may suit (I give the defining codes only):

Tiny	Ø, Ø, Ø, Ø, 16, Ø, Ø, Ø
	Ø, Ø, Ø, 24, 24, Ø, Ø, Ø
Medium	Ø, 16, 16, 124, 16, 16, Ø, Ø
	16, 16, 56, 254, 56, 16, 16, Ø
Large	146, 84, 56, 254, 56, 84, 146

4 (b) Some useful characters are:

the cross-hatched block		17Ø, 85, 17Ø, 85, 17Ø, 85, 17Ø, 85
the central blob		Ø, Ø, Ø, 24, 24, Ø, Ø, Ø
the diagonal lines		1, 2, 4, 8, 16, 32, 64, 128
	and	128, 64, 32, 16, 8, 4, 2, 1
pyramids	small	Ø, Ø, Ø, Ø, 24, 6Ø, 126, 255
	large	Ø, 24, 24, 6Ø, 6Ø, 126, 126, 255
central vertical		16, 16, 16, 16, 16, 16, 16, 16
central horizontal		Ø, Ø, Ø, 255, Ø, Ø, Ø, Ø

One could go on—use a square ruled exercise book to design, code and classify your graphics block codes. How many possible ones are there? Well, recall the tale of the merchant, the chessboard and the grains of rice. . . .

CHAPTER 7

1. Within the loop, use COLOUR 129, CLS, (delay), COLOUR 13Ø, CLS, (delay).
4. The main colours are 1–6 (Table 7.2), so set up a menu—"PRESS 1 for red", and so on. INPUT paper, COLOUR paper + 128, CLS etc.

CHAPTER 9

2. I guess actually the best way to make it friendly is to speed it up. And the best way to do *that* is to use '*integer variables*'. These can take whole number values only, and,

because the Electron doesn't have to mess about with decimal fractions, it can work a lot faster. About three times faster, in fact. Add

```
65  LET numb% = numb
```

and put % after each 'numb' and each 'go' in lines 70, 80 and 90. '%' after a variable name shows it's a numeric integer variable.

3. 'If you have time' is the crucial thing. I'd guess this program'd take many hours to run. But all it needs is an extra loop, FOR numb% = 1 TO 1E38 (or so) instead of the input, and a bit of messing around inside the 'go%' loop so it prints out 'go%' only if no factors are found.
4. You need to build your program round

   LET speed = 10*t and LET dist = 5*t.

   Who said physics is hard?
5. All I'll say is

   LET rage = 5 + 0.4 * (word/sent + poly/word)

   Why should I tell you in detail how to make your fortune?

CHAPTER 13

1. How about a simple

   ```
   45  IF LEN a$ < > 1 THEN GOTO 40
   ```

   Better to use REPEAT . . . UNTIL. Can you see how?
2. Base your work on the routine starting at 310 in Programs 16/17.

CHAPTER 14

1. The second task is the more simple—140 INPUT char1$: IF char1$ = "S" OR char1$ = "s" THEN . . . 145 INPUT char2$, char3$. PROCorder will need more thought.

Aren't I tantalizing?

CHAPTER 17

1. Best is to lay out the central and upper rows of letters as the white and black notes of a piano octave. Then you can use, for starters

```
REPEAT
LET g$ = GET$
IF g$ = "A" THEN PROCc              [Note C]
IF g$ = "W" THEN PROCcs             [Note C sharp]
. . .
IF g$ = "O" THEN PROCrest
```

Plenty of polish is possible to provide perfect playing (dig the alliteration!).

CHAPTER 20

1. Georgina will have to 'say' something like—'As I recall, your favourite drinks are this, that and the other. Would you like to change that list?' Then work up a PROCchange. Hic, hic, array.
2. This is an example of the important real-life process called *searching*. Use a loop:

   FOR item = 1 TO limit

   IF name$ (item) = target$ THEN PRINT " " ' name$,, number$: LET flag = 1 . . .

   NEXT item

   IF flag = Ø THEN PRINT ' ' ' "Not listed. Must be ex-directory."
3. As well as searching (question 2 above), you'll need to try some sorting (as in Program 2Ø). Sorting is the other major process of commercial computing.

APPENDIX 1

A3. I think the best thing to do is to make 'C' special, like 'S'. If the user presses C, use PROCcode which gives a new prompt, allows single-key entry of any number key (LET g = GET − 48), and prints a pre-defined character with PRINT CHR$ (224 + g). Alas, this is a bit clumsy, and in practice, you'll need to stick a crib-sheet back of the number keys, but it's the best the Elk can offer. Unless . . .
Oh, all right, you could explore entry 227 in the Appendix 3 section called 'Heavenly effects'.

Appendix 3: Tipples

Nothing to do with Bar-tender—that's my word for little tips. Sounds nicer than 'wrinkles', I think.

FULL SCREEN COLOUR CHANGE

Sometimes, for instance at the end of a program, you may wish to change the colour of a whole of an existing screen display. Here's the quickest way by far—it takes almost no time and looks lovely:

```
1ØØØ  VDU 19 paper new Ø;Ø
```

Here 'paper' is the existing background colour code, and 'new' is the code of the colour you want it to become.

Animating this effect is not so simple, however. We could get an 'explosion' by passing a few times through a loop of, for instance, VDU 19 128 7 Ø;Ø and VDU 19 128 Ø Ø;Ø. However, the TV's too slow for this and you need to cut down the speed with delays of about a fifth of a second between each colour change. Machine code is the only way to get real effectiveness here.

HEAVENLY EFFECTS

Which reminds me, have you stumbled on the curious case of the coloured listing? If you did, you perhaps got it (as I first did) by pressing CTRL + S instead of FUNC + S. After CTRL + S, a few stabs at certain keys (like Q or E or most numbers) livens up the screen no end. The answer is, CTRL + S is a form of VDU 19 . . .

But that's not what I really meant to look at here. 'Heavenly effects' describes the most useful of the so-called *FX calls, listed in the *User Guide* from Page 280.

Commands and program statements starting with a * involve calls to the Electron's 'operating system', most of which remains a closed book as far as we need to be concerned. You've used *KEY and *CAT by now, and may come across *LOAD or *CHAIN with certain commercial software; I'll also mention *OPT in a bit.

Anyway there's a series of *FX (= effects) calls you can sometimes use with effect (ho, ho). Here are the ones I think are the most important; I ignore others listed in the *User Guide* section mentioned above, and those involving work in assembly coding. Follow each with (R) if used as a direct command.

**FX Ø (R)* tells you which Acorn Electron operating system you have in your plastics box. Currently that's Version 1.00, but if Beeb practice is anything to go by, others will follow thick and fast. Then you'll need to know your OS, as software for later ones may not run on it. . . .

FX 4, 1 (comma, spaces and upper case optional as in all of these) blocks the usual use of the cursor control and COPY keys. They now print symbols—( ) + * '—and are available for use in games and sketching programs, for instance. Like this:

```
10  *FX 4 1
20  REPEAT: LET i$ = INKEY$(10)
30  IF i$ = "(" THEN PRINT "Left"
    . . .
70  UNTIL FALSE
```

FX 4 2 also blocks these keys' editing functions. This time you can program them in the same *KEY way as the digit keys and BREAK. Doubtful value to most of us, but I tell you anyway. Their key values are 11–15.

FX 4 0 undoes the effect of both the previous instructions.

FX 9 x and *FX 10 y* affect the flash rate of flashing colours, the values of x and y giving the time, in fiftieths of a second, for the first and second colours respectively. The *User Guide* is uncommonly clear about this.

FX 11 x and *FX 12 y* affect the delay before key repeat and the rate of repeat respectively. *FX 11 0* turns the repeat function off entirely; *FX 12 0* restores both to the normal values.

FX 15 1 empties the little bit of memory in which key presses are stored up. On the whole I use *FX 15 0* for this, though it has other similar but irrelevant effects. Try, for instance:

```
10  FOR a = 1 TO 100000
20  NEXT a
```

RUN, and while the program is RUNing type PRINT "secret message"(R). You won't see the letters on the screen, but when the looping is over your command will appear and be carried out. That won't work if you add *15 FX 15 1* (or 21 0), as this instruction empties that bit of memory.
The main uses are in cases (such as in Program A5) where you don't want earlier, so far unused, key presses to spoil the action of a program.

FX 18 empties the bits of memory holding your *KEY statements. (NEW and BREAK don't do that.)

FX 21 0 is like *FX 15 1* (as far as we are concerned).

FX 137 x turns the cassette motor on if x = 1 and off if x = 0. Jolly useful for program-driven audio cassette recordings.

FX 200, 1 stops the ESCAPE key working. This is often better than using ON ERROR . . ., as now ESCAPE has no effect, whereas the other way can't easily let the program keep going. Even more drastic is *FX 200, 2—whenever someone thereafter uses BREAK, the memory is cleared so well that not even OLD will bring back the program! This is a good trick if you don't want your programs listed, but not a good trick to tell people at school or all sorts of horrid practical jokes may result. Counter either effect with *FX 200, 0.

FX 201, 1 is also a practical joker's dream—it stops all the keys working except BREAK. *FX 201, 0 is the counter.

FX 202, 16 switches CAPS LK off if it's on. *FX 202, 0 does the reverse.

FX 210, 1 produces blessed silence as the sound system is switched off. A heavenly effect indeed! Bring it back if you want with *FX 210, 0.

FX 213, P changes the pitch of the beeps you get with VDU 7 (or CTRL + G) and pressing COPY when there's nothing to copy. P can be from 0 to 255 with the same values as in SOUND C, A, P, T. (*FX 214, T deals with duration T.)

FX 220, A lets you convert any key to ESCAPE by making A the ASCII value of the key concerned. So to effect "Press X to exit.", use *FX 220, 88. Then at any time a press of X will given an ESCAPE, while a press of ESCAPE won't.

FX 225–7 work on the FUNC function. The most useful uses are:

(a) *FX 225 prevents the user from using FUNC to get out your *KEY instructions.
(b) *FX 226 224 allows you to enter defined characters (the VDU 23 224 etc ones)

using FUNC + A (giving character 224), FUNC + B (character 225) and so on. Try this with Program A1(b) for instance. (If you have more than sixteen such character, use *FX 227, 240; that will put the characters 240 onwards on to the q, r, s and so on keys.)

KEY BLEEP

Non-brilliant typists like me need an audible indication of a key press. (Well, truly we need a sign that we've pressed the right key, but at the moment no micro can read minds.) Try the program in Figure . (It's worth saving on cassette for use before you start any coding.) Enter or LOAD the routine, without worrying how it works, RUN it, then type NEW to get rid of it. Get rid of it? Yes, but the effect lives on! Useful bit of machine code, that.

```
10  DIM PARA% 7: OSWORD=!(&20C) AND &FFFF
20  FOR i%=1TO8
30    READ p%:?(PARA%+i%-1)=p%
40    NEXT
50  DATA&11,0,&F6,&FF,&14,0,2,0
60  v%=!(&210)AND&FFFF
70  DIM P% 30
80  [
90  .beep PHA:TXA:PHA:TYA:PHA
100 LDA # 7
110 LDX # PARA% MOD 256
120 LDY # PARA%DIV256
130 JSR OSWORD
140 PLA:TAY:PLA:TAX:PLA
145 JSRv%
```

```
150 RTS
160 ]
170 ?(&210)=beep MOD 256:?(&211)=beep DIV 256
```

KEY PEEK

I haven't said much about the valuable keywords GET($) and INKEY($) in the body of the book, though I've used them on occasion, so here's a summary of their uses. All four check the keyboard for presses.

GET is a function returning the ASCII value of the key pressed next after it is called—and it'll wait for ever for a key press. You can use it rather like INPUT, but there's no annoying query-cursor, just the flashing underline, and the micro doesn't print the response on screen. Nor can it accept more than one key press. Here's the sort of thing it's good for:

```
1010 PRINT TAB (such and such) "Please press RETURN when you are ready
     to go on."
1020 REPEAT: UNTIL GET = 13                    [13 is RETURN key in ASCII]
```

If you are in a graphics mode, you can insert 1015 VDU 5 to get rid of the flashing cursor. If not, the best I can suggest is VDU 13, which moves the offensive object back to the start of the line.

GET$ is just the same except it returns a string containing the value of a key pressed. Thus line 1020 above could become REPEAT: UNTIL GET$ = CHR$ 13. Or you could have IF GET$ = "S" OR GET$ = "s" THEN STOP.

There's a great deal of sense in using one or other of the GET/INKEY family where the user need press only one key. (Indeed, so much sense is there, that it's worth encouraging this where you can with 'multiple-choice' menus and "press Y or N".) INSTR then gives a nice mug-trap routine. The sample assumes that the choice keys are A, B, C and H (for help):

```
100 LET g$ = GET$: ON INSTR ("aAbBcChH", g$) GOSUB 1100, 1100,
    1200, 1200, 1300, 1300, 9000, 9000 ELSE PRINT "Fool": GOTO 100
```

In the quotes go the acceptable responses; after GOSUB (or GOTO) put the addresses to which each response should send the micro; and then use ELSE to deal with the duds.

INKEY and INKEY$ are like GET and GET$ except they don't wait for ever. Put the allowed delay time, in hundredths of a second, in brackets after the keywords. So

```
IF INKEY$ 100 = "S" OR INKEY$ 100 = "s" THEN STOP
```

gives the hapless user one second to act correctly if (s)he wishes to stop! By the way, if there are a lot of key presses in your program, insert *FX 15 0 or *FX 21 0 before using any of the above. This wipes out any stray unused key presses from earlier.

MEMORY LEFT

Here's how to find how much memory space you've got left to fit the rest of your program into. Store the routine in a spare line (such as 0). Then you can access it when you like with G.0:

```
0 DIM P%-1: PRINT HIMEM-P%: STOP
```

Or put it on a FUNC key:

```
*KEY 5 DIM P%-1: PRINT HIMEM-P%|M
```

The % bit by the way, means that P is an integer variable. If variables are going to take only integer values, using % in their names makes the relevant lines work a bit faster and use less memory.

The quantity of spare memory you have depends on all kinds of things apart from the length of your current program. The main factor is the mode you're in (see table on Page 31), but variables take up space too, even if not used. See the effect on the above value if you enter DIM a (1000) for instance.

Sometimes you may need to know the number of bytes of memory your program listing takes up. Use PRINT TOP-PAGE for that.

RANDOM RESTORE

The instruction RESTORE n puts the data pointer at the first DATA item after line n. That leads to the possibility of randomising the order of questions posed in a simple test, for instance. The listing shows this very simply. Of course you can develop this to two or more DATA items (question/answer pairs) in each DATA line and build up nice long semi-random tests if you want to!

If you wish to block questions from being repeated, you need to use flag subroutines or arrays to hold your data. You may be able to work either method out from the details already given in the book.

```
10  LET ques = 0
50  REPEAT
60    RESTORE (99+INT(RND(5)+.5) ): LET ques = ques + 1: CLS:
      READ q$, a$: PRINT ' ' ' "Question "; ques; ":"
70    PRINT ' ' q$;: INPUT ans$
80    IF ans$ = a$ THEN PRINT ' "Correct!" ELSE PRINT ' "Nope!"
90    LET wait = GET: UNTIL FALSE
100 DATA What's your name, Becky
101 DATA Why smile, End of book
102 DATA On what day of the week did Suleiman the Tata first enter the
    Stamboul train, Dunno
103 DATA Why, Why not?
104 DATA What is NaCl, salt
105 DATA What is 10 + 10, 100
```

'RIGHT' AND 'WRONG' MESSAGES

It's rather dreary for a user to keep on getting the same micro reactions to correct or incorrect responses. Here's a (rather dreary too) way of randomizing them. (Dreary because some other micros can do the job much more simply.)

```
(pose question)
IF resp = ans THEN PROCyes ELSE PROCno
(and another question, etc)
...
DEF PROCyes
```

```
LET r = RND(4)
IF r = 1 THEN PRINT "Good."
IF r = 2 THEN PRINT "Right."
IF r = 3 THEN PRINT "Well done."
IF r = 4 THEN PRINT "Correct."
ENDPROC
DEF PROCno
(Well you get the idea)
ENDPROC
```

(Or you could use a random RESTORE as in the tipple called random RESTORE.)

Here's another way. It's not quite so dreary, and it's a neat example of string-handling. Can you suss it out? Note that all your alternate messages must have the same LEN.

```
DEF PROCyes
LET r = RND(4)-1: LET r$ = "Good!RightYes!!Super"
PRINT MID$(r$, 5 * r + 1.5)
ENDPROC
```

It's a shame we can't use: ON r PRINT "Good", "Right.", . . .

RNDRND

Where it is crucial that each time you run a program you get different values out of RND, put LET seed = RND(− TIME) somewhere close before the first use of RND. This will nicely randomize the random number set up. It won't work, though, if your Electron's been running continuously for more than about six months!

SAVINGS ACCOUNT

Here are some more useful tips for working with the cassette recorder.

Saving This is safest if several copies are made. Define one of the FUNC keys thus:

```
*KEY 9 SAVE "name" |M|M
```

and when you are ready to save your program press that key the desired number of times.

Verifying This means checking that the copy on tape exactly matches the original one in the micro's memory. It's very important to be able to do this, but alas it is not possible with the Electron. The worst thing to do is to try to reload the program you have just saved. While that is fine if there's no errors—if there are, you will end up with no program at all!

Instead use *CAT, which indicates on screen what is on the tape, and gives messages if there may be problems in some part of the recording.

Better still is to use *LOAD "name" 8000. (Or to get an even prettier effect—MODE 1: *LOAD "name" 4400.) This tries to send the tape material to part of the ROM, and this is impossible—however, the micro doesn't know that, so it tries, and will at least let you know of any major recording faults.

The ultimate verifying test is of course to load the recording into a second micro and compare the two listings. That's costly and time-consuming, but it *is* the best that the Electron can offer for verification.

Mind you, I don't wish to imply that saved programs are particularly unlikely to be properly recorded—but it does happen.

Loading Errors reported on the screen may sometimes be overcome by partial re-winding and reloading. If that fails, start loading again after typing *OPT 2 Ø. What this 'option' does is tell the micro to report errors it finds in loading, but otherwise ignore them—so it will load what it can, and let you try to work out the mess.

Merging This becomes more and more useful as you get on. It means loading in a program without wiping out the one that's there. The most important use is for when you wish to add a number of standard subroutines (or PROCedures) to a new program that calls on them. (A number of people find it useful to build up their own subroutine library, from which they can load relevant subroutines for use in a particular situation.)

The method for this is fairly straight-forward, in that it is normal to give subroutines and procedures line numbers in the thousands, well above those in the main program. Here's how to do it, to load a procedure called "3" from a cassette to add to memory on top of an existing program with lower line numbers.

```
OSCLI "LOAD" "3" " " + STR$(TOP-2)        (R)
```

When the material concerned has loaded, type END(R).

If the program on tape has line numbers that overlap those in memory (or lie below them), that method won't work. For such cases there's a much more hair-raising procedure that you can try; it's described on Page 200 of the *User Guide*.

Oh, by the way, to try to stop folks seeing your program listing (if that would embarrass you), chain it from a little starter program (also chained) like this:

```
10 PRINT TAB(Ø, 15) "The volume is correct: keep playing!"
20 CHAIN "name"
```

Then make the first two lines of your main program these:

```
Ø ON ERROR RUN                          [trap ESCAPE]
1 *KEY 1Ø 1 NEW |M RUN|M                [trap BREAK]
```

The only way out of that is to use CTRL + BREAK rather than BREAK; otherwise the program will not stop, or it will be lost.

SPEED LIMIT

Enter these two programs in turn, and time their RUNs.

```
  10 PROCsetup
  20 FOR counter = 1 TO 250
  30   REM  ** 1000 times loop
  40
  50   LET figure = PI*counter^2
  60   PROCcounteruse(counter, figure)
  70
  80   REM  ** End of loop coming up
  90   NEXT counter
 100 VDU 7
 110 END
 120
 130
 140
 150 DEF PROCsetup
 160 PRINT "Speed test program"'' "===== ==== ========"'''' "
Hope you remembered to time it!"
 170 ENDPROC
 180
```

```
190 DEF PROCcounteruse(number, area)
200 PRINT; number,, "£"; area
210 PRINT
220 ENDPROC
```

```
 10 PRINT"Speed test program"''"===== ==== ========"''''"Hop
e you remembered to time it!":REM*Loop now
 60 FORc%=1TO250:PROCc(c%,PI*c%*c%):NEXTc%
100 VDU7:END
200 DEFPROCc(n%,a%):PRINT';n%,,"£";a%:ENDPROC
```

I got the first one's time to be 14½ seconds; the second, the hare, reached its terminal beep in 12 seconds.
Here are the speed savers I used:

(a) Make the PROCedure called most often the one called first.
(b) Put PROCedures called only once into the main program instead.
(c) Minimize the number of spaces.
(d) Minimize the length of variable names.
(e) Use integer variables (with %) where you can.
(f) Minimize the number of REM-style lines.
(g) Avoid putting REMs within loops.
(h) Minimize the work to be done within loops.
(i) Use expressions rather than assignments (as in the PROC call).
(j) Avoid use of exponentiation where possible.
(k) Maximize use of multi-statement lines and PRINT structures.

The two programs do exactly the same thing. The second, the faster, also takes up far less memory.

The next version of the program is fully compacted in the above way—and takes just one line. It runs in two seconds now—but the main gain in speed results from my getting it to store the numbers in an array rather than printing them. The moral of that is—minimize screen output.

```
 10 DIMa(250):PRINT"Speed test program"''"===== ==== ======
="''''"Hope you remembered to time it!":FORc%=1TO250:LETa(c%
)=PI*c%*c%:NEXTc%:VDU7
```

But all this is getting ridiculous. Saving RUN-time and storage are valuable, but so too are structure and readability. Our speedy programs are not easy to read, and tend to be less well-structured. Trade-off needed.

A nice closing tip—to show up spaces in a listing, use the command VDU 23, 32, 255, 255, 255, 255, 255, 255, 255, 255 and LIST again. Using VDU 23 in the usual way, character 32—the space—is redefined as a white block. Any character can be redefined in this way—you can change the Electron's whole upper-case alphabet (for instance) to your own italic style if you've the urge. *Any* block of up to 32 consecutive characters can be dealt with using VDU 23 . . . A larger number than that needs an advanced technique.

TABULATION

A useful trick that I've used once or twice in this book already is to centre a string of unknown length on the screen. If the string is s$, use:

PRINT TAB(20−LEN s$/2) s$

This won't work if LEN s$ is greater than 40 of course, but you shouldn't use if for that anyway! The 40 figure, by the way, is the number of characters in the MODE 6 line. Change it where you need in other modes.

To tabulate strings neatly in a vertical list, the normal PRINT systems are quite

adequate. You can easily enough find a variation on the above system to tabulate them lined up at the right if you ever need that.

Lining up numbers is not so straightforward, however. Here's how to line numbers up with the decimal points underneath each other (in this example in column 25):

```
PRINT TAB(25-LEN STR$ INT number); number
```

The only case that this doesn't handle properly is when 'number' has a value less than 0.1. You may meet the problem quite often in certain kinds of program—it's an annoying Electron quirk! To get over it, beforehand use

```
LET @% = 121072 + 256*s,
```

where s is the number of decimal places you want.

WHILE

The Electron has two loop structures—FOR . . . TO . . . STEP . . . NEXT for use when you know how many times you're likely to want to loop, and REPEAT . . . UNTIL for when you don't. In the latter case, looping goes on until some specified condition changes as in this input routine:

```
10 LET go = 0
20 REPEAT
30   INPUT "What is 2 + 2", ans
40   LET go = go + 1
50   UNTIL ans = 4
60 PRINT ' ' "At last!"
```

A variation of REPEAT . . . UNTIL in some more advanced BASICs is WHILE . . . DO . . . END WHILE. WHILE . . . DO appears *comme ça*:

```
WHILE y < 0 DO
(set of instructions)
END WHILE
(next bit)
```

See? You can mirror it thus:

```
FOR go = 0 TO 1 STEP 0          [check that!]
(what to do)
IF y < 0 THEN NEXT go
(What to do now)
```

Or thus:

```
REPEAT
(so-and-so)
UNTIL y >= 0
(such and such)
```

OK?

Appendix 4: Resources

Note that at the time of writing (which I call 'currently' here), the Elk was a very new beast, negligibly available and with little support. By the time you read this, no doubt the situation will have changed drastically. The lists here are therefore, I am afraid, partly supposition.

USER GROUPS + PERIODICALS

ACORN USER c/o Addison-Wesley 53 Bedford Square London WC1 (Mainly BBC, monthly.)
ELECTRON USER 68 Chester Road Hazel Grove Stockport (Currently a supplement to the excellent MICRO USER, a Beeb-specific monthly.)
MAPE 18 Cuttings Avenue Sutton Notts (An important association for primary teachers using micros, that will, I hope, take much note of the Electron.)
MICRONET Orton Centre Peterborough (The main closed user group on Prestel, currently giving much support to the BBC micro and planning to work with the Electron. If it succeeds, it will be a super source of news and programs through the telephone system.)
MICRO USER 68 Chester Road Hazel Grove Stockport (A superb glossy BBC micro monthly.)
MUSE PO Box 43 231/2 Friary Chambers Whitefriargate, Hull HU1 2HD (The main educational computing association, with its own magazine—the marvellously edited *Computers in Schools*—and a huge library of learning software. Very likely to support the Electron.)
ORBIT PO Box 50 St Albans Herts (an offshoot of the huge Beebug, a BBC user group with tens of thousands of members. Orbit plans software development and a monthly magazine, but currently has somewhat delayed its start-up.)
PERSONAL COMPUTER NEWS A weekly bookshop glossy, the one most likely to be of value to Electron users.

BOOKS

You have three now—this one, and the *User Guide* and Yazdani's little book that come with the machine. Currently three others have appeared:
Deeson, E *Learning with the Elk* (AVC Software PO Box 415 Harborne Birmingham B17 9TT £2. (£1.50 for members of Beebug, MUSE and Orbit)) Ten programs of use with children of most ages, with programming notes and index.
Ferguson, J & Shaw, T *Assembly language programming on the Electron* (Addison-Wesley, £7.95) A good coverage of the subject (but *not* applicable to interfacing as the blurb claims).

McGregor, J & Watt, A *The Electron Book* (Addison-Wesley £7.95) A detailed, but often hard-going coverage of BASIC written in fact for the BBC and rather hastily converted.

Other books will doubtless appear with speed. On the basis of previous products, I would expect those from Granada, Macmillan, Prentice-Hall and Shiva to be best value.

SOFTWARE

Currently software has appeared from only three sources:
Acornsoft (the beginnings of a doubtless extensive range of good but pricey products)
AVC Software PO Box 415 Harborne Birmingham B17 9TT (programs for learning in many areas at primary and secondary level)
Orbit PO Box 50 St Albans Herts (currently only a rival to the Introductory Cassette, but I expect more of value to be added)

A number of companies, large and small, are known to have plans for this market, including A + F, Collins, Heinemann and Micro Power. You will need to investigate local suppliers of the micro, if any, and/or scan the advertisements in the magazines (with the wariness usually expected in dealings with small mail-order companies). If you can get to a home computing exhibition/fair, you are likely to be able to view a good range of products.

HARDWARE

Currently no 'add-on' electronic products for the Electron have appeared. Look out for interfaces allowing use of printers (cheap or costly), disc storage systems, and data capture/process control. But beware, this can make yours an expensive hobby! Companies that may become important here include Griffin & George, Sir Computers and Watford Electronics.

Index

abbreviations, 36, 48
ABS, 55, 142, 143
accumulator, 123
Acornsoft, 161
ACS, 55
action, 70
Addison-Wesley, 160, 161
ALU, 9
amplitude, 107, 108
AND, 66, 94, 131
animation, 24, 82
apostrophe, 20, 22
argument, 55
arithmetic, 35, 40, 51, 52, 96
arithmetic/logic unit, 9
arrays, 95, 117, 118, 120, 130, 132, 155, 158
arrow keys, 25, 35, 36, 46, 50
ASC, 96, 138, 146
ASCII, 94, 96, 97, 100, 118, 153, 154
ASN, 55
assembler, 11, 33, 34, 121
assignment, 13, 15, 21, 29, 34, 62, 119, 158
Atherton, 90
AT, 34
ATN, 55
attack, 107, 108, 111
AUTO, 12, 13, 18, 19, 40, 44, 49, 50, 110
AVC Software, 2, 160, 161

background, 13, 15, 37, 41, 136, 147
back-space, 96
back-up store, 9
bar-chart, 135
BASIC, 1, 10, 20, 33, 40, 63, 121
BBC micro, 1, 11, 33, 58, 108, 111, 120, 124
BGET, 121
binary, 47, 51, 52
bit, 47
BPUT, 121
brackets, 19, 35, 52, 53, 66, 80
BREAK, 12, 13, 18, 24, 37, 45, 54, 122, 125, 128, 152, 153
BREAK-proofing, 12, 54, 157
buffers, 110, 152, 154
bugs, 16, 49, 82, 90, 101
byte, 47, 124

calculator, 7, 51
CALL, 34, 122
capitals, 15, 17, 19, 35, 93
CAPS LK, 4, 11, 15, 17, 18, 36, 45, 48, 49, 50, 93, 109, 111, 153
cassette control, 121, 123, 152
cassettes, 5, 9, 11, 34, 58, 70, 120, 156
catalogue, 60
CHAIN, 10, 44, 49, 58, 59, 120
channels, 107, 108, 109, 110, 112
characters, 11, 18, 21, 25, 29, 31, 34, 92, 94, 102, 117, 137, 146, 148
chips, 33
CHR$, 25, 29, 96, 98
circle, 144
CLG, 128, 130
click, 108, 153
CLOSE, 121
CLS, 5, 10, 13, 21, 37, 96, 125, 130
code, 25, 94, 96, 100
coding, 16, 33, 87
colour, 13, 31, 37, 40, 51, 124, 126, 127, 148, 151
COLOUR, 5, 10, 13, 15, 23, 34, 37, 40, 49, 90, 101, 127, 130, 147
comma, 21, 41, 50, 51, 118, 119, 121
commands, 5, 10, 11, 12, 34, 44, 114
comment, 31, 115
Commodore, 34
computed address, 30, 77
computer, 7, 51
concatenation, 29, 82, 93, 96, 137, 140, 141, 142
constants, 22, 23, 29, 92
control, 123
  unit, 9
COPY, 36, 46, 50, 152, 153
COS, 55, 141
counting, 19, 24, 62
CTRL, 25, 27, 31, 36, 45, 54, 78, 96, 151, 157
cursors, 4, 5, 36, 46, 102, 126, 128, 152, 154

dangers, 4, 16
data, 8, 47, 117, 121
DATA, 92, 117, 118, 146
data processing, 8, 51
debugging, see bugs
decimal fractions, 54
decisions, 62, 70, 74
dedicated micro, 7
DEF, 40, 56, 78, 98
default, 127
DEG, 55, 111
delay, 24, 30, 38, 40, 49, 103, 138, 143, 154
DELETE, 44, 49, 114
  key, 20, 36, 50
dialects, 33, 68
difference, 55
digital micro, 8
DIM, 95, 117, 122, 123, 132, 154
disc, 120, 121
display, 18, 19, 21, 22
DIV, 55
divide, 35, 51
DIY, 25, 34, 137, 148, 158
documentation, 70
DRAW, 125
dual processing, 109, 110
duration, 107, 108, 110, 119

editing, 19, 36, 45, 46, 53, 103, 114, 152
effects, 112, 138, 151
efficiency, 16, 48, 157
Electron User, 161
element, 132
ELSE, 34, 55, 64, 65, 84, 119, 131, 142, 154
END, 34, 40, 48, 78, 88, 109, 126
English, 17, 33
ENVELOPE, 108, 110
EOF, 121
equations, 53
errors, 11, 12, 16, 19, 30, 35, 36, 44, 49, 52, 53, 60, 101, 116, 118, 131

ERROR, 53, 60, 106, 118, 126, 128, 152
ESCAPE, 12, 13, 24, 31, 36, 38, 40, 44, 53, 61, 106, 122, 126, 128, 153, 157
exponentiation, 21, 35, 40, 51, 53, 158
expression, 13, 52

factors, 56
FALSE, 28, 39, 40, 48, 62, 99
files, 34, 35, 120
flags, 88, 116, 133, 140, 141, 155
flash, 152
floppy tape, 121
flowcharts, 14, 16, 39, 63, 65, 67, 68, 73, 74, 84, 85, 104, 105
FN, 56, 81, 98
FOR, 38, 48, 62, 84, 101
foreground, 13, 15, 37, 41, 136, 147
format, 18, 19, 20, 21, 22, 51, 102
frequency, 107
functions, 35, 52, 54, 88, 96
FUNC, see CAPS LK

gaining, 25, 119, 137, 141, 142, 143, 144, 145, 152
GCOL, 130, 138
general-purpose micro, 7
GET($), 34, 89, 97, 98, 99, 102, 103, 117, 131, 133, 149, 154
GOSUB, 74, 76, 79, 84, 90, 115
GOTO, 30, 34, 41, 45, 48, 63, 76, 90, 102, 115
graphics, 25, 27, 29, 31, 34, 124, 137, 145, 152
graphs, 130, 135, 143
Griffin & George, 161
guarantee, 4, 107

hash, 121

IF, 15, 41, 62, 84
indent, 19, 48
initialization, 87, 145
ink, see foreground
INKEY($), 34, 97, 98, 99, 112, 138, 143, 154
input, 9, 15, 70
INPUT, 13, 15, 19, 20, 21, 22, 30, 34, 48, 49, 51, 67, 76, 92, 95, 97, 116, 117, 121, 126, 134, 154
INSTR, 97, 154
INT, 54, 96, 99
integer variables, 122, 123, 148, 155, 158
interaction, 25, 117, 123
interface, 123
Introductory Cassette, 4, 5, 7, 9, 10, 58, 60, 161
item, print, 20

JMP, 122, 123
JSR, 122, 123

keyboard, 153
keyword, 12, 17, 25, 34, 35, 36, 48, 64, 93
kilobyte, 47

label, 123
language, 20, 33
LDA, 122, 123
LEFT$, 97, 99, 120
LEN, 76, 96, 97, 99, 100, 117, 131
LET, 13, 15, 19, 21, 24, 29, 34, 62, 64, 119, 122, 134
library, 5, 70
line numbers, 10, 12, 19, 36, 44, 46, 63, 79, 90, 114, 147
LIST, 12, 19, 31, 36, 44, 48, 60, 151, 157
LISTO, 45, 50, 136
LOAD, 11, 45, 48, 58, 59, 120, 157
LOCAL, 79, 99
LOG, 35, 55
logic, 66, 92, 94, 127, 139, 141
LOGO, 33
loop, 12, 13, 15, 38, 41, 43, 45, 62, 101, 133, 158, 159
loudness, 107
lower case, 15, 17, 19, 35, 93

machine code, 33, 34, 48, 121, 151, 153
magazines, 160
main-frame, 8
MAPE, 160
maths, 17, 21, 35, 40, 51, 89, 92, 100, 131, 136, 139, 141, 142, 143
memory, 16, 31, 47, 48, 72, 87, 93, 102, 118, 119, 124, 132, 134, 154, 155, 158
menu, 30, 141, 154
messages, 17, 18, 22, 41, 49, 87, 90, 116
micro-electronics, 8
Micronet, 160
Micro User, 161
MID$, 97, 99, 117, 120, 142, 146, 156
mistake, 60
MOD, 55
MODE, 5, 10, 13, 24, 34, 37, 42, 98, 125, 136
modes, 13, 18, 21, 24, 31, 87, 90, 111, 124, 127, 131, 137, 155
module, 16, 74, 84, 90, 101
motor control, 121, 123, 152
MOVE, 126, 143, 145
mug-trap, 30, 77, 88, 98, 116, 117, 131, 133,
multiple statements, 49, 60, 64, 65, 103, 158
multiply, 35, 51, 53
MUSE, 160

names, 35, 59, 77, 80, 88, 93, 135, 158
nesting, 66, 74, 76, 79, 135
NEW, 12, 15, 18, 45, 114, 152
NEXT, 38, 48, 102
nibble, 47
noise, 112
NOT, 94
numeric, 13, 21, 22, 29, 39, 134, 159

OFF, 60, 102
OLD, 12, 18, 45, 48, 50, 114, 152
ON, 53, 60, 77, 102, 115, 156
OPENIN, 121
OPENOUT, 121
OPENUP, 121
operating system, 47, 48, 151
Orbit, 160, 161
OR, 66, 94, 133
origin, 125, 130
outline, 70, 72
output, 9, 15, 18, 70
overheating, 59, 60

page, 45, 87
paper, see background
parameters, 80
pattern, 19, 23, 28, 42, 50, 130, 136
PEEK, 34, 48, 146
peripherals, 9, 34, 45
Personal Computer News, 161
PET, 134
PI, 56, 141
picture, 23, 29
pitch, 107, 108, 119
planning, 16, 84, 102
PLOT, 128, 130, 145
pointer, 118
POKE, 34
polishing, 90
powers, 21, 35, 40, 51, 53
power supply, 3, 4, 6, 45, 59
precedence, see priority
prime numbers, 56

PRINT, 5, 10, 13, 15, 18, 19, 22, 24, 29, 38, 48, 49, 52, 102, 121, 126, 158
printer, 9, 11, 24, 34, 35
priority, 52, 53, 66, 109
PROC, 40, 78, 80, 110, 133
procedures, 31, 35, 40, 56, 78, 84, 90, 98, 106, 125, 137, 158
processor, 9
program, 5, 7, 9, 10, 11, 59
programmed keys, see *KEY
  characters, see DIY
programming, 16, 20, 33, 48, 70, 71, 84, 90, 101, 132, 147
prompt, 4, 5, 11, 13, 15, 21, 46, 49

quality, 107

RAD, 55, 111
radian, 55
RAM, 48
random, see RND
random walk, 24
READ, 118, 134, 140
recording, see cassettes
REM, 31, 40, 43, 46, 48, 71, 76, 78, 103, 115, 158
RENUMBER, 45, 48, 112, 119, 148
repeat, key, 50, 152
REPEAT, 13, 15, 19, 21, 22, 23, 24, 28, 34, 38, 40, 48, 62, 84, 136
REPORT, 106
resolution, 31
re-starting, 15, 30, 85, 117, 118
RESTORE, 118, 155, 156
RETURN, 74, 82
  key, 5, 10, 11, 15, 44, 49, 50, 116, 154
RIGHT$, 97, 120
RND, 13, 24, 28, 30, 34, 55, 99, 101, 103, 110, 118, 119, 126, 137, 144, 155, 156
rogue value, 15, 23, 57, 85
ROM, 48
routine, 16
RUN, 11, 12, 13, 18, 34, 37, 45, 46, 59, 78

SAVE, 11, 46, 48, 59, 156
scale, 131
screen layout, see display
scroll, 36, 48
search, 150
semi-colon, 19, 20, 38, 40, 41, 93
set, character, 25, 94, 100, 118
SGN, 55
SHIFT, 5, 12, 13, 15, 17, 31, 36, 45
SIN, 35, 55
Sinclair, 1, 3, 4, 34, 36
Sir Computers, 161
sorting, 95, 150
sound, 4, 24, 96, 107, 153
SOUND, 34, 108, 119, 136, 142, 144
spaces, 11, 13, 18, 35, 45, 48, 119, 158
SPC, 19, 29, 88, 95, 134
speaker, 107, 111
speed, 8, 16, 43, 103, 121, 123, 124, 148, 157
SQR, 53, 55, 96
STEP, 39, 48, 159
statement, 5, 10, 12, 20, 44, 49
stop, 24, 28, 126
STOP, 71, 76, 77, 109
storage, 8, 9, 11, 47, 62, 95, 117
story, 131
STR$, 98, 100, 131
strings, 19, 21, 24, 29, 35, 82, 88, 92, 117, 119, 135, 142, 146, 156
STRING$, 27, 29, 96, 99, 140, 142, 146
structure, 31, 35, 80, 84, 90, 158
sub-routine, 16, 31, 35, 36, 74, 84, 101, 122, 123
sub-strings, 25, 36, 97
subscript, 132
symbols, 15, 35, 68, 94
syntax, 12, 60, 101, 116
synthesizer, 108, 109

TAB, 19, 22, 24, 28, 30, 34, 40, 98, 99, 100, 119, 124, 131, 158
TAN, 55, 141
tape, see cassettes
testing, 84, 90, 101
THEN, 15, 62, 64
tilde, 25
timbre, 107
time, 59, 97, 107, 138
TO, 38
top-down development, 16, 90, 101, 106
tracing, 102
TRACE, 102
triangles, 129
TRUE, 62
TV, 3, 4, 5, 6, 8, 9, 11, 18, 59, 107, 125

underline, 30, 35, 40, 50, 53
UNTIL, 49, 99, and see REPEAT
upper case, 15, 17, 19, 35, 93
user, 9, 17, 71
  -friendly, 17, 18, 21, 56, 71, 85, 90, 115, 116, 148, 154, 155
  groups, 160
  Guide, 2, 4, 5, 24, 60, 111

VAL, 88, 98, 99, 116
variables, 13, 20, 22, 29, 35, 38, 45, 50, 79, 80, 88, 93, 110, 158
VDU, 49, 96, 118
VDU5, 126, 128, 137, 154
VDU7, 24, 117, 136, 153
VDU14, 95, 130
VDU19, 41, 95, 98, 118, 119, 126, 127, 128, 130, 151
VDU23, 25, 34, 101, 111, 119, 140, 158
VDU24, 133
VDU29, 130
verify, 60, 156
volume, 107

Watford Electronics, 161
WHILE, 62, 159
window, 130
word, 47
working area, 3, 6

Yazdani, 4, 5, 160

zero, 11, 24, 25, 39, 52
zodiac, 29, 31
zones, 21, 124

£, 121

*, 12, 35, 51, 115, 151
*CAT, 60, 151, 156
*CHAIN, 151
*FX, 109, 138, 140, 143, 151
*KEY, 12, 18, 50, 54, 78, 108, 109, 110, 120, 151, 153, 154, 156
*LOAD, 58, 151, 156
*OPT, 151, 157
*RUN, 58, 151

< = > etc, 13, 15, 94
?, 34, 48, 146
@, 159
', 12, 50, 78

If you are at the 'L' stage with your Electron then take some instruction the easy way!

Starting from scratch, the author leads you lightheartedly but effectively through the intricacies of computing. The fundamental topics of animation, strings, arrays and sound are covered with over 25 sample programs reproduced directly from bug-free printouts.

There are also complete chapters on the important techniques of flowcharting, debugging and editing.

Another 16 programs are listed and ready to RUN at the back of the book, including:

- Target practice
- Lunarvader
- Hangman
- Snake in the grass

Electrify your friends with your programming skills with this latest title from Eric Deeson.

GB £ NET +005.95

ISBN 0-906812-66-6

00595

9 780906 812662

SHIVA

Shiva Publishing Limited

Easy Programming for the Electron

Eric Deeson

SHIVA